(1)

A Specification of the Soar Cognitive Architecture in Z

Brian G. Milnes
with contributions from
Garrett Pelton, Robert Doorenbos, Mike Hucka,
John Laird, Paul Rosenbloom and Allen Newell

August 31, 1992

CMU-CS-92-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

B. Milnes, G. Pelton, B. Doorenbos may be reached at the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890. A. Newell is deceased, but his academic estate may also be reached at the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890. M. Hucka and J. Laird may be reached at the Electrical Engineering and Computer Science Department, The University of Michigan, Ann Arbor, MI 48109-2122. P. Rosenbloom may be reached at the Information Sciences Institute and Department of Computer Science, University of Southern California, 4676 Admiralty Way, Marina del Rey, CA 90292.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

Keywords: Artificial Intelligence, formal specification, integrated intelligent architectures, machine learning, problem solving, Z specifications, Z notation, Soar.

Abstract

A formal specification of the sixth revision of the Soar architecture in the Z notation was constructed to elucidate and clarify the definition of Soar and to guide its implementation. Soar is a cognitive architecture that has been successfully applied to many domains and has been proposed as an exemplar unified theory of cognition. Z is a model theoretic specification language based in set theory that has syntax and type checking programs available. The specification has a complete coverage of the architecture, a low level of abstraction and a considerable implementation bias.

For Allen Newell (1927-1992) — who taught me that computer science is an empirical quantitative discipline.

—B.G.M.

Contents

List of Figures

Chapter 1

Introduction

1.1 An Introduction and the Intended Audience

A formal specification of the sixth revision of the Soar cognitive architecture [LNR87] was constructed in the Z specification notation [Spi89, Spi88]. Soar is a cognitive architecture that has exhibited a wide variety of performance in domains such as: machine learning [LRN84], cognitive modeling [Aky90, DS90, JRS91], development [SK91], natural language processing [LLN91], expert systems [HPS89] and robotics [LYHT91].

This specification was written to guide the construction of implementations of Soar and to facilitate the understanding and extension of the Soar architecture. Formal specification has simultaneously been applied to two other related computional architectures by Iain Craig of the University of Warwick, UK [Cra91]. Craig used Z to specify blackboard architectures and his Cassandra architecture and similarly applied the specification to guide Casandra's implementation.

The Z notations is introduced as it is used. It is largely founded upon familiar set theory, but most readers will be better served by first reading an introductory text, such as [PTS91, Wor92, Hay85], and the Z manual [Spi89].

Artificial intelligence researchers who have an interest in understanding the details of Soar will find many answers to their questions here. Indeed, the process of writing the specification has made clear many details of the architecture to its implementors and founders. Considerable knowledge of the Soar architecture from less formal sources, such as [LCAS90], would ease and enhance the reader's understanding. Z specifiers will most likely find this difficult reading, but we do draw some conclusions about the strengths and weaknesses of using Z for a large, detailed, low-level specification of a computational architecture.

1.2 The History and Philosophy of the Specification

In January 1990 we began an effort to re-implement the current Soar architecture, intending only a few well understood modifications. As we proceeded, it quickly became clear that the limiting factor was our understanding of the details of the architecture itself. Although Soar's architecture description papers [LNR87, LRN86] and manual [LCAS90] described some of the architecture, they were out of date with the current system (Soar 5.2), at too high a level of abstraction and had obvious inconsistencies with the system.

In an earlier revision of Soar (4.2) a small module (preference semantics) had been formalized in first order logic [Lai86]. When we updated and reorganized this formalization, we discovered that it had a pleasing clarity and disambiguity that allowed us to quickly answer precise questions about the module's operation.

After considerable reflection, we decided to undertake a formal specification of the basic functionality of the architecture. Our hope is that this document will become the definitional standard of Soar. That is, the specification's terms will become the language that we use to discuss the architecture. And, when we change the architecture, we will first update the specification to flush out our understanding of the changes, and then distribute the changed specification for comment. This way, the detailed content of the architecture will not remain hidden in the implementation and become obscured with time.

After an initial, unsuccessful attempt to formally specify the architecture in an ad hoc language, we undertook a small survey of the popular specification languages. Our intuition was that a model theoretic language would serve the purpose of specifying a cognitive architecture better than an algebraic system such as Larch [Gut90]. We looked at Z, VDM [Jon90, JS90], and OBJ [GW88], but chose Z. Z appealed most to us because of its strengths in composing small specifications into larger specifications, the availability of good programs to textually format, syntax and type check its specifications and the local expertise here at Carnegie Mellon with Z [Gar91, DG89].

Although Z has proved a very capable specification language it lacks two properties that would have considerably simplified our task. First, Z's lack of a full execution semantics forced us to use a cumbersome state machine notation. Second, although we agree that not all specifications written in a language should be executable, if Z was directly interpretable it would have allowed us to much more quickly debug the specification.

1.3 The Specification's Coverage and Implementation Bias

The specification is intended to cover all the functionality of what we view as "the Soar architecture", and avoid details that are specific to the implementation. Initially we felt that the ideal specification would be a very abstract specification of the architecture with little implementation bias because an abstract specification would simplify the modification of the specification and help to prevent discourse on the design from being overwhelmed in details.

Several problems quickly derailed our aspirations for an abstract specification. We wanted the specification to guide our re-implementation of Soar. We desired an easily instrumented and flexibly modified implementation. This drove us to produce a more detailed specification that could be used as an index into the implementation. Also, as we have never had any specification of the architecture that was more concrete than general prose but less concrete than code, we found it difficult to produce abstract specifications that had sufficient detail to guide the implementation.

The result is a specification that abstracts away from many implementation details but is a very detailed accounting of the operation of the system. Some implementation oriented details, such as techniques for inserting "hooks" into the system, Soar's use of numbers, and a conjunctive match condition ordering algorithm have been included in appendixes. We leave the problem of how to produce a more abstract specification to future Soar researchers as they attempt to understand, modify and implement new revisions of the architecture.

1.4 An Overview of the Architecture

Figure 1.1 diagrams a module and control oriented view of Soar, and Figure 1.2 diagrams a module and data oriented view. Soar is composed of four modules and is enclosed in an interface, which we call the external interface. Figure 1.1 represents each module by a rectangle, each state machine control by a circle or oval, and the calling hierarchy by arrows. Figure 1.2 represents each module in a rectangle, each state machine with a circle or oval, each memory with a heavy lined rectangle, and the reading and writing of each memory with arrows.

The External Interface Soar is designed as a situated autonomous agent that directly interacts with the external world, but an actual implementation of the architecture is a program that allows a user to run Soar as if it is an embedded agent. We distinguish Soar the program, from Soar the architecture, by wrapping Soar in the external interface. This interface contains the user interface that allow users to view, control and modify the execution of Soar. The architecture provides only one operation to the external interface, *StepTL*.

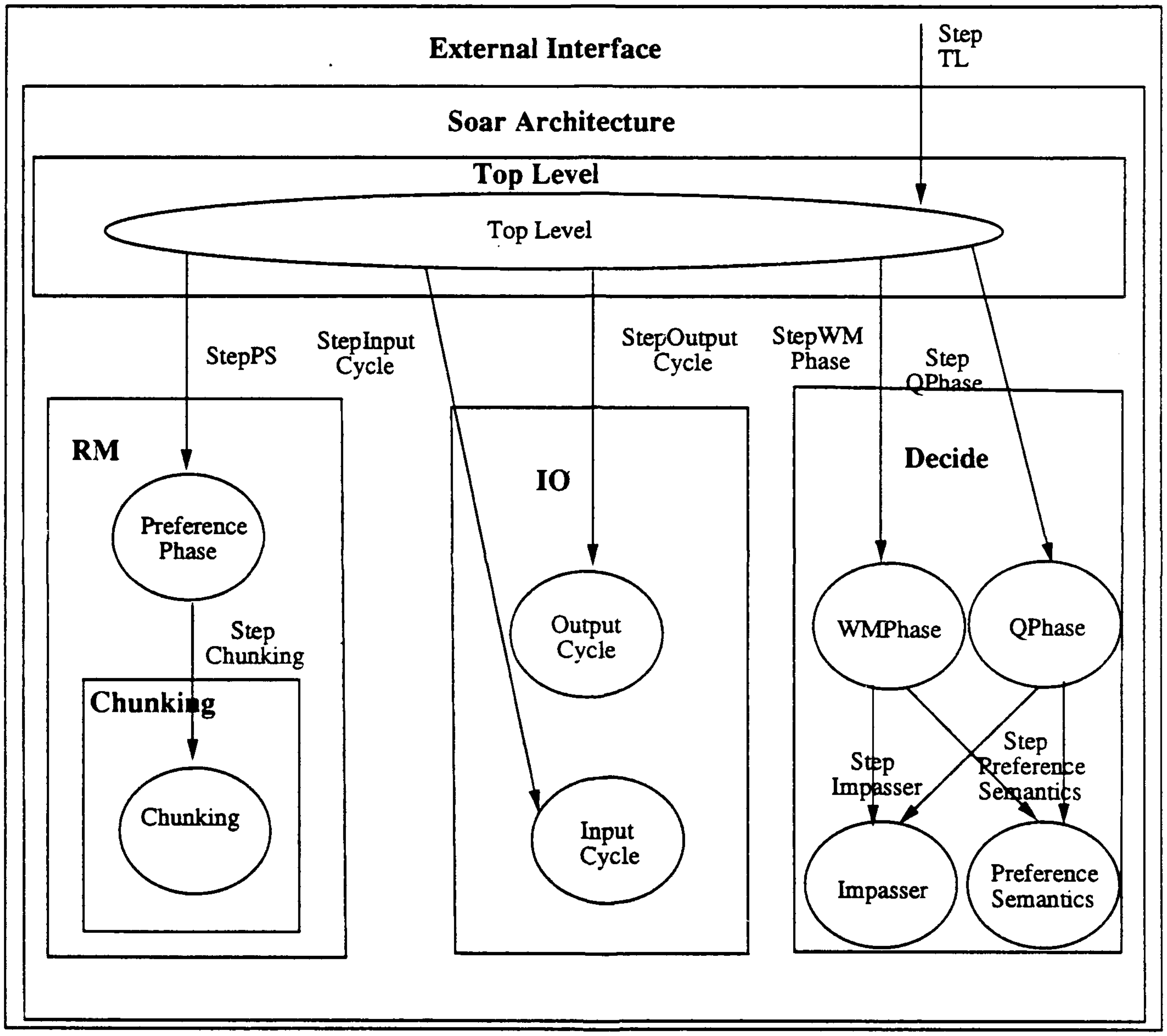

Figure 1.1: Module and Control Block Diagram

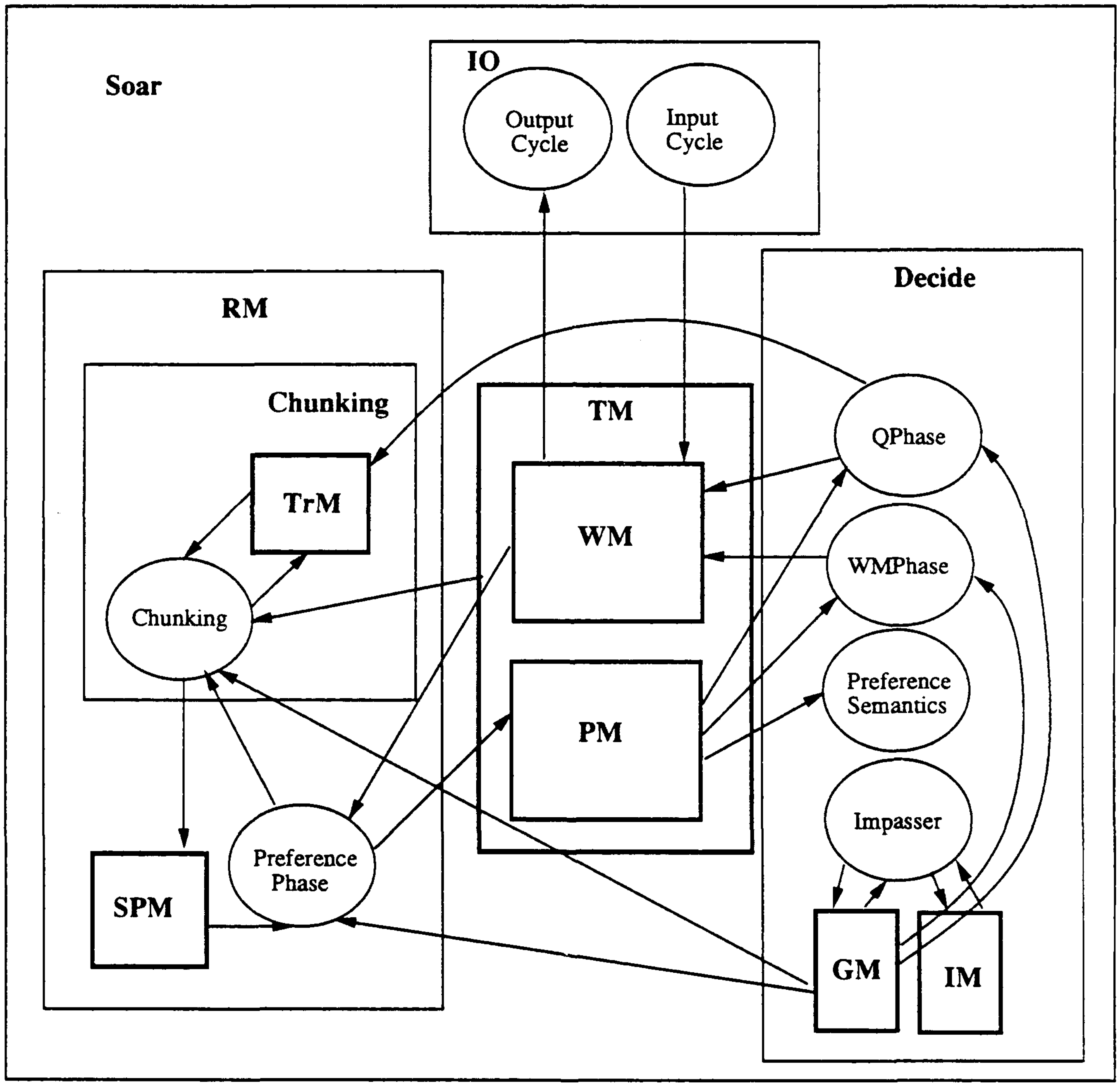

Figure 1.2: Module and Data Block Diagram

StepTL moves Soar one step forward. It acts much like a clock line of a CPU that an engineer externally pulses.

Soar's four modules are: the top level, recognition memory (RM) including chunking, input output (IO) and the decision procedure (Decide).

Recognition Memory Recognition memory is Soar's memory model; it contains a long term memory in the form of production rules called production memory, or *SPM*. Its short term memory is temporary memory, *TM*, which is split into two parts called working memory, *WM*, and preference memory, *PM*. As all of the modules of Soar read and write temporary memory, Figure 1.2 depicts it as global to the entire system. The preference phase is a loop that recalls information from long term memory until no new recollections are available.

Chunking Chunking is Soar's learning mechanism. It observes the changes to temporary memory and preference phase's recollections and stores them in two trace memories (*TrM*). From these and Decide's goal stack, it learns new memories and adds them to production memory. The external interface steps the top level, which calls *StepPS* to step recognition memory's preference phase, which calls *StepChunking* to step the chunking module.

IO The IO module allows Soar to communicate with the outside world. The top level steps IO's input and output cycles with *StepInputCycle* and *StepOutputCycle*. The input cycle reads perceptions from input channels, and adds them to temporary memory. The output cycle finds the subsets of temporary memory that represent motor actions and ships them out output channels to the outside world.

IO requires transducers to map Soar's motor commands into a form that simulated or real motor controllers can understand, and to map the perceptions of simulated or real perceptual devices into Soar's form. These transducers are housed in the external interface, and are also viewed as being part of SoarIO, but not part of the architecture.

Decide The decision procedure is Soar's universal subgoaling mechanism. It maintains two goal memories, *GM* and *IM*, and interprets the contents of preference memory as requests to change the contents of pieces of working memory. Whenever a set of requested changes are not consistent, or do not provide a unique change to working memory, decide generates a new goal to represent the problem. The decision procedure has two state machines: working memory phase (WMPhase) and quiescence phase (QPhase), stepped by *StepWMPhase* and *StepQPhase*. The working memory phase and the quiescence phase both step the impasser and preference semantics, with *StepImpasser* and *StepPreferencePhase*.

1.5 An Overview of the Specification

The specification is organized into eight chapters and is supplemented with four appendixes on implementation related considerations.

1. Chapter 1, Introduction — this introduction and overview of the specification.
2. Chapter 2, Specifying Soar's Control: State Machines — introduces the state machine convention for specifying the high level control of Soar.
3. Chapter 3, Base Symbol Structures — introduces the basic symbol types which Soar composes into more complicated symbol structures.
4. Chapter 4, Temporary Memory and Production Memory — defines the temporary memory and production memory data structures that are used throughout the specification.
5. Chapter 5, Recognition Memory — defines the operation of the recognition memory, including Soar's learning mechanism, Chunking.
6. Chapter 6, IO — specifies Soar's mechanism for communicating with the external world.
7. Chapter 7, Decide — define's Soar's universal subgoaling mechanism.
8. Chapter 8, Top Level — specifies the top level control loop that orchestrates the execution of Soar's modules.
9. Appendixes —
   (a) Appendix A, Numbers — specifies how to augment the base symbol structures of Soar with a single type of numeric type.
   (b) Appendix B, An Implementation Discipline — describes techniques for creating an implementation that is very close to the specification.
   (c) Appendix C, Finer Grained Hooks — describes a technique that would allow an implementation to observe and modify the implementation at a finer granularity than would otherwise be supported by the specification.
   (d) Appendix D, The Reorderer — specifies a greedy heuristic reordering algorithm to sort the conditions of individual Soar productions into an order that on average matches faster.

Chapters 2, 3 and 4 cover introductory structure and are required to understand the remaining chapters. Chapters 5, 6, 7 and 8 cover the details of the major components of Soar, and may be read in any order. Although the Z specification language requires a bottom-up definitional approach. chapters 5. 7 and 8 refer forward in prose and backwards in Z and prose. Two readings of these chapters is probably required to clearly understand their complicated interactions.

Chapter 2

Specifying Soar's Control: State Machines

This chapter describes how the structure of Z and the desired properties of the design and implementation of Soar conspire to produce many constraints on the specification of Soar. Section 2.1 discusses the alternative representations of control in Z. Section 2.2 describes the technique adopted to handle this problem: state machines.

2.1 Specifying Soar's Control

Z's technique for modeling a computational system is called a "sequential system model" ([Spi89] 130-141). Unfortunately, Z's sequential system model does not provide sufficiently rich execution semantics. As Soar is a computational model of cognition, a large part of its design is its flow of control. Although a specification that abstracts away from Soar's detailed control could prove useful, to guide our implementation effort, we would prefer that our specification carefully specify Soar's flow of control.

We have investigated three alternative techniques for specifying Soar's control in Z: prose, execution sequences and state machines.

We began by specifying pieces of the architecture's flow of control in prose. This prose described how individual state changes were combined in sequences and loops. However, it lacked clarity, coverage and precision.

Next, we experimented with specifying control by defining execution sequences. We constructed the set of all sequences of states of Soar, and then constrained them to start with an initial state and only follow defined state changes. However, as Soar contains several components each of which contains isolated control, this quickly became cumbersome. Finally, we settled upon a state machine control paradigm.

The state machine control makes explicit the legal sequencing of operations in the Soar model by using state diagrams, a Z representation of the states of the machine and a Z sequential operation for each transition of the state machine. The individual transitions of the state machine are composed into a single operation that moves the machine forward one of the possible transitions with each application.

In this chapter we construct a prototype state machine format that allows state machines to:

- initialize
- take one step of a possible set of transitions
- step an embedded sub-state machine
- determine when a state machine has finished executing and
- reset

The individual state machines that control parts of Soar are constructed following the form of this prototype, and then composed into a single machine that drives all of Soar. The resulting implementation of the architecture is controlled from an interface that we call the external interface. This interface drives Soar step by step, much like an engineer externally pulses the clock of a computer processor.

2.2 The Prototype State Machine

This section defines the state machine control conventions by exhibiting a state machine prototype. The prototype state machine is diagrammed in Figure 2.1. The states of the machine are diagrammed with ovals, and the operations that move the system between states are diagrammed as arrows.

The Names of the States The names of the states of the machine are represented in Z using the free type *StateCounter*. In this case, *StateCounter* is defined to be a type that is a set of five distinct variables named: *SMInitialState*, *SMS1State*, *SMS2State*, *SMS3State*, and *SMFinishedState*.

$$\begin{aligned} StateCounter ::= {} & SMInitialState \mid SMS1State \mid SMS2State \mid \\ & SMS3State \mid SMFinishedState \end{aligned}$$

Free types are Z's method for constructing disjoint unions ([Spi89] 81). They are similar to union types in C ([KR88] 147), but are mathematically motivated ([BW90] 258). In this case, the free type definition introduces only new variables so it acts more like a C enumeration ([KR88] 39).

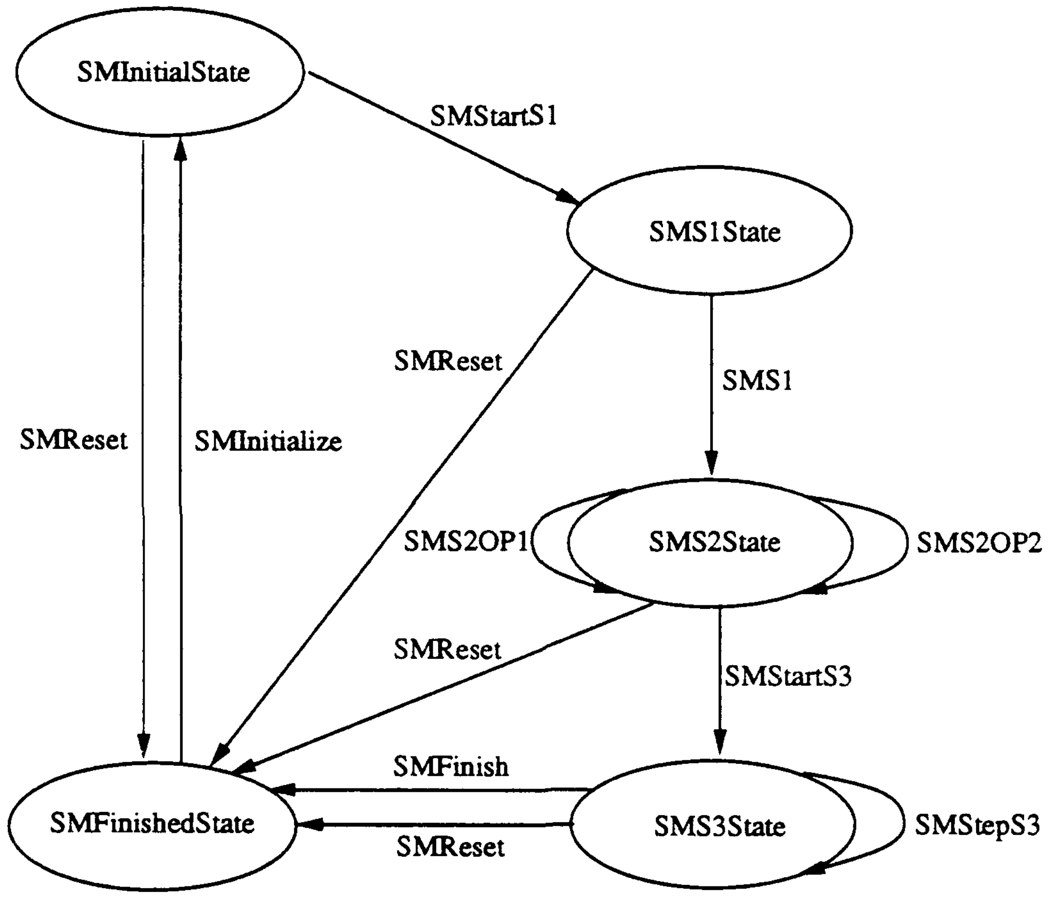

Figure 2.1: State Machine Prototype

Z defines the free type notation by expansion into an equivalent Z basic type definition ([Spi89] 50) with a set of new variables of disjoint values. In the first part of this expansion, *StateCounter* is introduced as a basic type; e.g., a set of values with no predetermined structure.

$[StateCounter]$

An axiomatic description is then used to introduce the five new variables. The part of the axiomatic definition above the bar is called the declaration or signature. The part below the bar is called the predicate. The declaration introduces the five new variables, typing them as elements of the set *StateCounter*. The scope of variables defined in an axiomatic description is everything after the bar in the rest of the specification. The predicate part defines predicates on the variables introduced by the declaration. In this case, the built-in Z predicate *disjoint* is applied to a sequence of the singleton sets of the variables to constrain the values to which they are bound to be distinct.

$$\begin{array}{|l}
SMInitialState, SMS1State, SMS2State, SMS3State, \\
\quad SMFinishedState : StateCounter \\
\hline
disjoint\langle \{SMInitialState\}, \{SMS1State\}, \{SMS2State\}, \{SMS3State\}, \\
\qquad \{SMFinishedState\}\rangle
\end{array}$$

The Functions of the States The five different states each perform unique functions in the prototype state machine.

1. *SMFinishedState* — when the machine is not being used, it rests in this state. The driver for this machine calls an initialize action to move it into the *SMInitialState*.
2. *SMInitialState* — this is the state from which the machine begins computation.
3. *SMS1State* — in this state the *SMS1* operation is applied to the state.
4. *SMS2State* — in this state the *SMS2OP1* or *SMS2OP2* operations are applied to the state, or *SMStartS3* is applied, moving the machine to state *S3*.
5. *SMS3State* — in this state the *SMStepS3* operation is applied to step a sub-state machine. When the sub-state machine has finished, the *SMFinish* operation ends the execution of the state machine.

Z's sequential system model uses schemas to specify almost all of the parts of a computational model. A schema can specify a data structure, like a record structure in a programming language. A schema can represent the environment (or state) of a computational model, like the declaration of global variables in a programming language. A schema can specify predicates on an environment, like a boolean function in a program or a program invariant. A schema can also specify an operation that changes the environment, like a procedure in a programming language. Perhaps Z's greatest strength is its schema calculus. The calculus provides a rich and uniform way to combine schemas that are playing any of these roles.

The Total State of the Machine The prototype state machine's computational state is represented here using a schema named *StateSchema* ([Spi89] 51). Like an axiomatic declaration, the part above the bar is a declaration and below the bar is a predicate. The scope of variables defined in a schema is only the text from the bar to the end of the schema box. The scope of the name of the schema is all the specification past the bottom bar of the schema definition. The prototype's state schema defines only *state_counter* to remember the state the machine currently occupies. In the specification of an actual Soar state machine, the declaration's ellipses (...) would declare other variables to enrich the state model, and the predicate's ellipses would define state invariants.

StateSchema
$state\_counter : StateCounter$
...
―――――
...
―――――

Initial States for the Machine The Z convention to define a predicate on the initial states of a sequential system model is to prefix the name of the state schema with "*Init*" ([Spi89] 131). Thus the *InitStateSchema* defines a predicate which initial states of the prototype state machine must satisfy. The declaration part of this schema references the schema *StateSchema*. Z defines this reference to mean that the new schema contains all of the declarations of the referenced schema and all of the referenced schemas's predicates are conjoined with the new schema's predicate ([Spi89] 53).

InitStateSchema
StateSchema
―――――
$state\_counter = SMFinishedState$
...
―――――

The expanded definition of *InitStateSchema*, shown below, is that of a schema with the same signature as the state schema, but the state counter is constrained to start in the finished state. Again, the ellipses is used to signify that the initial state schemas of state machines might contain other restrictions.

```
— InitStateSchema ————————————————
  state_counter : StateCounter
  . . .
  ————————
  state_counter = SMFinishedState

  . . .
——————————————————————————————————
```

The General Form of Operations After defining the form for the state of our sequential system model and constrained its initial values, we must define operations that move the system through its state space. Operations that change the state of the sequential system are called sequential operations ([Spi89] 130). A sequential operation is a relation on pairs of states. Z uses relations instead of functions to allow the specification of non-deterministic operations. The specification takes good advantage of this: for example, decide non-deterministically selects between indifferent operator candidates by using an non-deterministic sequential operation schema.

For a schema to define a relation between two states it must have a way to reference the component variables of both states. Z's technique is to construct a schema that references both the state schema and a copy of the state schema in which all of the components have been decorated (post appended) with an single quote (′) ([Spi89] 32). The undecorated state variables refer to the start state and the decorated components refer to the resultant state For example, the *SMOperation* schema defines all of the components of StateSchema and a decorated copy of the components.

```
— SMOperation ————————————————————————
  state_counter, state_counter′ : StateCounter
  ————————

——————————————————————————————————————
```

Decorating all of the components of a schema is common enough that decorating the schema name is defined to decorate all of its components. For example, *SMOperation* could be defined by referencing both a decorated and an undecorated copy of the state schema.

```
__ SMOperation ________________________________
 StateSchema
 StateSchema'
______________
 . . .
_______________________________________________
```

However, sequential operations are so common that a second short hand, Δ, is defined to mean the schema and a decorated copy of the schema.

```
__ SMOperation ________________________________
 ΔStateSchema
______________
 . . .
_______________________________________________
```

The Form of a State Transition Now that we have defined the state of the system, its initial states and the general form of a state changing operation, we need to define the form of typical state machine operations. When a state machine's caller wants to initialize the system, it calls an initialization operation. Operations of this form pairs state schemas with counters in the finished state to state schemas with counters in the initial state.

```
__ SMInitialize _______________________________
 ΔStateSchema
______________
 state_counter = SMFinishedState

 state_counter' = SMInitialState

 . . .
_______________________________________________
```

Start Transitions When a state machine can loop in a state, the operation that starts the loop by moving to the state is named with "*Start*".

$$
\begin{array}{|l}
\hline
\textit{SMStartS}1 \\
\Delta \textit{StateSchema} \\
\hline
\textit{state\_counter} = \textit{SMInitialState} \\
\ldots \\
\textit{state\_counter}' = \textit{SMS}1\textit{State} \\
\hline
\end{array}
$$

Simple Transitions The *SMS1* operation provides the form for the simplest type of operation. It performs its one and only change to the state, represented by ellipses. At the same step it moves the state machine into the next state, *SMS2*.

$$
\begin{array}{|l}
\hline
\textit{SMS}1 \\
\Delta \textit{StateSchema} \\
\hline
\textit{state\_counter} = \textit{SMS}1\textit{State} \\
\ldots \\
\textit{state\_counter}' = \textit{SMS}2\textit{State} \\
\hline
\end{array}
$$

Looping and Compound Transitions *SMS2* style operations are more complex than the *SMS1* style operation. At each step the machine can change the sequential system state by either the *SMS2OP1* operation or the *SMS2OP2* operation. After the application of these operations, the machine remains in *SMS2State* allowing *SMS2* to loop until *SMS2OP1* and *SMS2OP2* are both no longer applicable.

$$\begin{array}{|l}
\hline
SMS2OP1 \\
\Delta StateSchema \\
\hline
state\_counter = SMS2State = state\_counter' \\
\ldots \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
SMS2OP2 \\
\Delta StateSchema \\
\hline
state\_counter = SMS2State = state\_counter' \\
\ldots \\
\hline
\end{array}$$

We would like to be able to define an operation that allows the state machine to take either an SMS2OP1 or an SMS2OP2 transition. The Z schema calculus ([Spi89] 74) allows the convenient combination of schemas by extending the standard logical operations of and ($\wedge$), or ($\vee$) and not ($\neg$) to apply to schemas. Two schemas may be combined with $\wedge$ or $\vee$ if their signatures give all shared variables the same types. The signature of the resulting schema contains all of the signature elements of both of the schemas and conjoins or disjuncts the predicates of the two schemas. Any schema may be negated: the resulting schema has the same signature and the negation of the predicate.

$$SMS2 \mathrel{\widehat{=}} SMS2OP1 \vee SMS2OP2$$

The *SMS2* operation is defined as an "or" of the *SMS2OP1* transition and the *SMS2OP2* transitions. The $\widehat{=}$ operation names a new schema to be the value of a schema calculus expression. This new transition satisfies the conditions of either *SMS2OP1* or *SMS2OP2*. The *SMS2* operation schema does not actually need to be defined; the machine works identically with just the *SMS2OP1* and the *SMS2OP2* transitions installed. (We define such transitions ***only*** to give the users of Soar an easy way to observe related operations: such as all the operations that leave a state, see Appendix B.) The state diagrams will not have an item representing the compound transitions because they are difficult to draw: some cases only require one arrow with two different names, but some would require a graphic merging of arrows.

In a sequential system model not all operations are applicable to all states.

The schema calculus provides an operation, called pre-condition (pre) to calculate if an operation is applicable ([Spi89] 72). The pre-condition operation maps a sequential operation to a predicate that checks if all of the conditions on the initial state of the operation are satisfied. Essentially, it tests a schema for membership in the domain of the transition relation.

SMStartS3 checks that *SMS2* is no longer applicable by using the pre-condition of *SMS2*. When *SMS2* is no longer applicable, *SMStartS3* moves the state into *SMS3State*. This exit transition makes *SMS2State* a while-do looping state: it applies *SMS2* until it is no longer applicable and then exits.

SMStartS3

$\Delta StateSchema$

$state\_counter = SMS2State$

$\neg\ \mathrm{pre}\ SMS2$

$InitializeSubStateMachine$

$state\_counter' = SMS3State$

SubState Machine Transitions Soar is too complicated to specify with just one state machine. We need to compose machines such that one state machine can step another state machine. *SMS3* prototypes a state that is implemented by a sub-state machine, named SubStateMachine. When *SMStartS3* applies, it initializes the sub-state machine using *InitializeSubStateMachine*. *SMStepS3* loops in the *SMS3State* and applies the *StepSubStateMachine* to step the sub-state machine.

SMStepS3

$\Delta StateSchema$

$state\_counter = SMS3State = state\_counter'$

$StepSubStateMachine$

$SMS3 \mathrel{\widehat{=}} SMStartS3 \lor SMStepS3$

SMFinish checks that the sub-state machine is finished by checking the precondition of *StepSubStateMachine*, and moves the state machine into its *SMFinishedState*.

$$
\begin{array}{|l}
\textit{SMFinish} \\
\hline
\Delta \textit{StateSchema} \\
\hline
\textit{state\_counter} = \textit{SMS3State} \\
\neg\ \mathrm{pre}\ \textit{StepSubStateMachine} \\
\ldots \\
\textit{state\_counter}' = \textit{SMFinishedState} \\
\hline
\end{array}
$$

Stepping a Machine The caller of a state machine requires a single compound operation to step a sub-machine. *SMStep* is a prototype of an operation to step an entire machine forward one step. The stepper allows the machine to move through one step of any operation except its initialization.

$$\textit{SMStep} \mathrel{\widehat{=}} \textit{SMStartS1} \lor \textit{SMS1} \lor \textit{SMS2} \lor \textit{SMS3} \lor \textit{SMFinish}$$

The *SMReset* operation resets the state machine to the finished state. It is designed to be called when the state machine's execution is interrupted, and would be drawn as an arrow from every state to the finished state. As it is also cumbersome to draw this, it will not be shown in the other state machine diagrams. In an implementation, *SMReset* is designed to take actions to recover the state of the implementation that would otherwise be lost by an initialize operation.

$$
\begin{array}{|l}
\textit{SMReset} \\
\hline
\Delta \textit{StateSchema} \\
\hline
\textit{state\_counter}' = \textit{SMFinishedState} \\
\ldots \\
\hline
\end{array}
$$

The prototype state machine provides a convention for representing complicated control, a single step at a time, using Z's weak state relation sequential operation semantics. The convention provides templates for:

- initialization
- start transitions
- simple transitions
- looping and compound transitions

- substate machine transitions
- finishing transitions and
- machine stepping transitions.

The specification defines each piece of Soar's control flow in a state machine, and composes them into a single operation that moves Soar one atomic control step.

Chapter 3

Base Symbol Structures

This chapter defines the base sets of symbols that Soar uses to construct its knowledge representations. Soar uses two base set of symbols: Σ and the *SpecialSymbol* set. Σ is a set of base general symbols for knowledge representation. Soar composes these symbols to construct all of its knowledge structures. *SpecialSymbols* are the reserved words of Soar. They are symbols with meanings specific to the operation of Soar.

$$[\Sigma, \mathit{SpecialSymbol}]$$

Soar partitions Σ into the sets of variable symbols, identifier symbols, and constant symbols. The three subsets of Σ are typed as elements of $\mathbb{P}\,\Sigma$, the set of all subsets of Σ ([Spi89] 27). The Z partition predicate relates a sequence of sets to a set if and only if the sets in the sequence are disjoint and their union is equal to the set ([Spi89] 125).

$$\begin{array}{|l}
\mathit{Variable}, \mathit{Identifier}, \mathit{Constant} : \mathbb{P}\,\Sigma \\
\hline
\langle \mathit{Variable}, \mathit{Identifier}, \mathit{Constant} \rangle \text{ partition } \Sigma
\end{array}$$

As many definitions allow either variables or constants in their domain we define *VariableOrConstant* for brevity.

$$\begin{array}{|l}
\mathit{VariableOrConstant} : \mathbb{P}\,\Sigma \\
\hline
\mathit{VariableOrConstant} = \mathit{Variable} \cup \mathit{Constant}
\end{array}$$

Also many definitions use either an identifier or a constant in their domain. Soar users commonly refer to these as *Symbols*.

$$\begin{array}{|l}
\mathit{Symbol} : \mathbb{P}\,\Sigma \\
\hline
\mathit{Symbol} = \mathit{Identifier} \cup \mathit{Constant}
\end{array}$$

The *SpecialSymbols* are partitioned into *PreferenceSymbols* and *RelationSymbols*. The *PreferenceSymbols* appear in the preference slot of preferences. The *RelationSymbols* appear in the syntax of relational tests inside of productions. Typographically, we use single quoted variable names to emphasize that these are constants of Soar, but to Z they are just Z variables. As the symbol $`='$ is Soar's name for both the equality relation and the indifferent preference, to we distinguish them with subscripts of "r" for relation and "p" for preference.

$$
\begin{array}{|l}
\mathit{PreferenceSymbol}, \mathit{RelationSymbol} : \mathbb{P}\ \mathit{SpecialSymbol} \\
`@', `!', `+', `-', `\sim', `>', `<', `=_P', `\&' : \mathit{SpecialSymbol} \\
`=_R', `<>', `<=>' : \mathit{SpecialSymbol} \\
\hline
\langle \mathit{PreferenceSymbol}, \mathit{RelationSymbol} \rangle \text{ partition } \mathit{SpecialSymbol} \\
\mathit{RelationSymbol} = \{`=_R', `<>', `<=>'\} \\
\mathit{PreferenceSymbol} = \\
\quad \{`@', `!', `-', `-', `\sim', `>', `<', `=_P', `\&'\}
\end{array}
$$

The decision procedure's input language is the set of preferences. Preferences describe properties of the structure of Soar's short term memory. The preferences may describe a property of a single object in memory, called unary, or may describe a property of two objects in memory, called binary. The sets *UnaryPreferenceSymbol* and *BinaryPreferenceSymbol* partition the preference symbols into those that can appear in unary preferences and those that can appear in binary preferences. All of the binary preference symbols are also unary, but some of the unary preferences (such as $`-'$) can not appear in binary preferences.

$$
\begin{array}{|l}
\mathit{UnaryPreferenceSymbol}, \mathit{BinaryPreferenceSymbol} : \mathbb{P}\ \mathit{PreferenceSymbol} \\
\hline
\mathit{UnaryPreferenceSymbol} = \\
\quad \{`@', `!', `+', `-', `\sim', `>', `<', `=_P', `\&'\} \\
\mathit{BinaryPreferenceSymbol} = \{`>', `<', `\&', `=_P'\}
\end{array}
$$

The specification of the Soar architecture references 17 constant symbols. The *disjoint* predicate ensures that the variables all take no distinct values from the set of constants. No specific identifiers or variables are defined because Soar uses these as placeholders that describe the structure of its knowledge representations, but not the specific content.

```
| `NIL', `T', `GOAL', `PROBLEM-SPACE', `STATE', `OPERATOR',
| `IMPASSE', `OBJECT', `ITEM', `ATTRIBUTE', `CHOICES',
| `CONSTRAINT-FAILURE', `CONFLICT', `TIE', `NO-CHANGE',
| `NONE', `MULTIPLE', `QUIESCENCE', `TYPE' : Constant
|-----------
| disjoint ⟨{`NIL'}, {`T'}, {`GOAL'}, {`PROBLEM-SPACE'},
|     {`STATE'}, {`OPERATOR'}, {`IMPASSE'},
|     {`OBJECT'}, {`ITEM'}, {`ATTRIBUTE'},
|     {`CHOICES'}, {`CONSTRAINT-FAILURE'}, {`CONFLICT'},
|     {`TIE'}, {`NO-CHANGE'}, {`NONE'},
|     {`MULTIPLE'}, {`QUIESCENCE'}, {`T'},
|     {`TYPE'}⟩
```

Chapter 4

Temporary Memory and Production Memory

This chapter specifies the elements of two of Soar's memories: temporary memory (TM) and production memory (PM). Temporary memory holds TMEs, a disjoint union type constructed from *Preferences* and *WMEs*. The preferences are also stored in preference memory (PM), and the working memory elements in working memory (WM). Temporary memory is the union of preference and working memory. Production memory holds productions that match against working memory, instantiate, fire and retract to modify preference memory.

4.1 Preferences

Soar's preferences are the input language of the decision procedure. Decide reads preference memory to determine how to change working memory. Soar has two types of preferences: unary and binary.

A Unary preference describe a property of an identifier, attribute, value triple. The most common property is that a value is acceptable for installation in WM for an identifier, attribute combination, called a slot.

```
__ UnaryPreference ____________________________
| id : Identifier
| attribute, value : Symbol
| preference : UnaryPreferenceSymbol
|______________________________________________
```

We must de-structure the data elements contained in top level memories to specify general data invariants of Soar, such as the set of all identifiers currently in use anywhere within Soar. *Components* mappings are used to de-structure an element of some type into the set of its basic symbols.

The *UnaryPreferencesComponents* mapping extracts the identifier, attribute and value symbols from a unary preference.

$$UnaryPreferencesComponents : UnaryPreference \rightarrow \mathbb{P}\,\Sigma$$

$$\forall u : UnaryPreference \bullet$$
$$UnaryPreferencesComponents(u) = \{u.id, u.attribute, u.value\}$$

Binary preferences describe a property of two values for a slot to Decide. A common property described is that the identifier, attribute and value triple is a better choice for installation in working memory than the identifier, attribute and referent triple.

BinaryPreference

$id : Identifier$
$attribute, value : Symbol$
$preference : BinaryPreferenceSymbol$
$referent : Symbol$

The *BinaryPreferencesComponents* mapping extracts the identifier, attribute, value and referent symbols from a preference.

$$BinaryPreferencesComponents : BinaryPreference \rightarrow \mathbb{P}\,\Sigma$$

$$\forall b : BinaryPreference \bullet$$
$$BinaryPreferencesComponents(b) =$$
$$\{b.id, b.attribute, b.value, b.referent\}$$

The preference free type constructs the disjoint union of unary preferences and binary preferences. The *UnaryP* and *BinaryP* total injections ([Spi89] 106) are used to construct a *Preference* from a unary or binary preference.

$$Preference ::= UnaryP\langle\langle UnaryPreference\rangle\rangle \mid BinaryP\langle\langle BinaryPreference\rangle\rangle$$

Given a preference, you can check if it is constructed from a unary preference or a binary preference by testing it for membership in the range of the injectors $p \in \operatorname{ran} UnaryP$ or $p \in \operatorname{ran} BinaryP$ ([Spi89] 9). The inverse functions, $UnaryP^{\sim}$ or $BinaryP^{\sim}$ can then be used to unpack a preference into its unary or binary preference ([Spi89] 101). *PreferencesComponents* de-structures a preference using this technique.

$$PreferencesComponents : Preference \rightarrow \mathbb{P}\ Symbol$$

$$\begin{array}{l}
\forall\, p : Preference \bullet \\
\quad ((p \in \operatorname{ran} UnaryP \Rightarrow \\
\qquad PreferencesComponents(p) = \\
\qquad\quad UnaryPreferencesComponents(UnaryP^{\sim}(p))) \land \\
\quad (p \in \operatorname{ran} BinaryP \Rightarrow \\
\qquad PreferencesComponents(p) = \\
\qquad\quad BinaryPreferencesComponents(BinaryP^{\sim}(p))))
\end{array}$$

The specification frequently creates preferences, so two functions are provided to map the components of preferences to a preference. The μ operator acts much like a programming language's let construct and the θ operator instantiates a schema using the values for its components provided in the current scope ([Spi89] 61, 64).

$$\begin{array}{l}
makeUnaryPreference : \\
\quad Identifier \times Symbol \times Symbol \times UnaryPreferenceSymbol \\
\qquad \rightarrow Preference \\
makeBinaryPreference : \\
\quad Identifier \times Symbol \times Symbol \times BinaryPreferenceSymbol \times Symbol \\
\qquad \rightarrow Preference
\end{array}$$

$$\begin{array}{l}
\forall\, i : Identifier;\ a, v : Symbol;\ p : UnaryPreferenceSymbol \bullet \\
\quad makeUnaryPreference(i, a, v, p) = \\
\qquad UnaryP((\mu\ UnaryPreference \mid \\
\qquad\qquad id = i \land attribute = a \land value = v \land preference = p \bullet \\
\qquad\quad \theta\, UnaryPreference)) \\
\\
\forall\, i : Identifier;\ a, v, r : Symbol;\ p : BinaryPreferenceSymbol \bullet \\
\quad makeBinaryPreference(i, a, v, p, r) = \\
\qquad BinaryP((\mu\ BinaryPreference \mid \\
\qquad\qquad id = i \land attribute = a \land value = v \land \\
\qquad\qquad preference = p \land referent = r \bullet \\
\qquad\quad \theta\, BinaryPreference))
\end{array}$$

PreferencesId is defined to allow easy access to the identifier slot of both unary and binary preferences.

$$PreferencesId : Preference \rightarrow Identifier$$

$$\begin{array}{l}
\forall\, p : Preference \bullet \\
\quad ((p \in \text{ran}\ UnaryP \Rightarrow PreferencesId(p) = (UnaryP^{\sim}(p)).id) \wedge \\
\quad\ (p \in \text{ran}\ BinaryP \Rightarrow PreferencesId(p) = (BinaryP^{\sim}(p)).id))
\end{array}$$

The architecture specifies that only preferences which are accessible from the most recent goal, or any impasse object, should remain in preference memory. We start the formal definition of accessible with the concept of connecting identifiers over a universe set of preferences. $Connected_{Preference}$ is a mapping that takes a set of preferences to a relation between identifiers. Two identifiers are related over a set of preferences if there exists a preference that holds the first identifier in its id slot and the second in its value or referent slot.

$$Connected_{Preference} : \mathbb{P}\ Preference \rightarrow (Identifier \leftrightarrow Identifier)$$

$$\begin{array}{l}
\forall\, P : \mathbb{P}\ Preference;\ l, r : Identifier \bullet \\
\quad (l, r) \in (Connected_{Preference}(P)) \Leftrightarrow \\
\quad\quad (\exists\, p : P \bullet \\
\quad\quad\quad ((p \in \text{ran}\ UnaryP \Rightarrow \\
\quad\quad\quad\quad (UnaryP^{\sim}(p)).id = l \wedge (UnaryP^{\sim}(p)).value = r) \wedge \\
\quad\quad\quad (p \in \text{ran}\ BinaryP \Rightarrow \\
\quad\quad\quad\quad (\exists\, b : BinaryPreference \mid b = BinaryP^{\sim}(p) \bullet \\
\quad\quad\quad\quad\quad b.id = l \wedge (b.value = r \vee b.referent = r)))))
\end{array}$$

The connection formed is roughly analogous to a directed edge in a graph; here each unary preference would be viewed as an attribute labeled edge from id to value, and each binary preference would be viewed as constructing two attribute labeled edges from id to value and from id to referent. However, the correspondence is not very natural because the node set of a graph of preferences would also have to contain all of the constants. Soar allows only identifiers to be augmented with a preference (e.g., appear in the id slot of a preference) and so the graph would require the additional constraint that only identifier nodes have out edges.

$TCI_{Preference}$ extends the concept of a single preference connecting pairs of identifiers to a path of preferences connecting sets of identifiers. The * operation calculates the reflexive transitive closure of the relation $Connected_{Preference}(P)$ ([Spi89] 102). The closure is then imaged ([Spi89] 101) over all of the identifiers in the start set. This produces an operation that finds all of the identifiers connected to any identifier in the start set through a path of preferences. The syntax $\_(\!|\_|\!)$ is used to denote the image of a function on a set of arguments ([Spi89] 101). The image of a function on a set, S, is the set of the function applied to all of the elements of S.

$$\underline{TCI_{Preference}} : \mathbb{P}\ Preference \rightarrow (\mathbb{P}\ Identifier \rightarrow \mathbb{P}\ Identifier)$$

$$\forall P : \mathbb{P}\ Preference;\ I : \mathbb{P}\ Identifier \bullet$$
$$(TCI_{Preference}(P))(I) = ((Connected_{Preference}(P))^*)(\!|I|\!)$$

$OfId_{Preference}$ extracts from a set of preferences those preferences that hold in their id component any element of the given set of identifiers.

$$\underline{OfId_{Preference}} : \mathbb{P}\ Identifier \times \mathbb{P}\ Preference \rightarrow \mathbb{P}\ Preference$$

$$\forall I : \mathbb{P}\ Identifier;\ P : \mathbb{P}\ Preference \bullet$$
$$OfId_{Preference}(I, P) = \{p : P \mid PreferencesId(p) \in I\}$$

4.2 Working Memory Elements

Soar's working memory is a set of elements named working memory elements. Working memory elements have an id component, and an attribute and value components. They are of two types: those that are a copy of an acceptable preference for a context slot and those that are not. The *context_acceptable_preference* component holds a flag of the free type, *YesOrNo* to distinguish these two types.

$YesOrNo ::= Yes \mid No$

```
── WME ──────────────────────────────
  id : Identifier
  attribute, value : Symbol
  context_acceptable_preference : YesOrNo
─────────────────────────────────────
```

The components of a working memory element is the set of its identifier, attribute and value.

$$\begin{array}{|l} WMEsComponents : WME \rightarrow \mathbb{P}\,\Sigma \\ \hline \forall w : WME \bullet WMEsComponents(w) = \{w.id, w.attribute, w.value\} \end{array}$$

The *makeWME* constructor is defined to simplify the instantiation of working memory element schemas.

$$\begin{array}{|l} makeWME : Identifier \times Symbol \times Symbol \times YesOrNo \rightarrow WME \\ \hline \forall i : Identifier;\ a, v : Symbol;\ c : YesOrNo \bullet \\ \quad makeWME(i, a, v, c) = \\ \quad\quad (\mu\ WME \mid id = i \land attribute = a \land value = v \land \\ \quad\quad\quad context\_acceptable\_preference = c \bullet \\ \quad\quad\ \theta\, WME) \end{array}$$

$Connected_{WME}$ defines the notion of connectedness of identifiers across a single working memory element from a universe set of working memory elements.

$$\begin{array}{|l} Connected_{WME} : \mathbb{P}\ WME \rightarrow (Identifier \leftrightarrow Identifier) \\ \hline \forall W : \mathbb{P}\ WME;\ l, r : Identifier \bullet \\ \quad (l, r) \in (Connected_{WME}(W)) \Leftrightarrow \\ \quad\quad (\exists\, w : W \bullet w.id = l \land w.value = r) \end{array}$$

TCI_{WME} extends the connectedness definition, in the same way $TCI_{Preference}$ did, to include a set of identifiers and any path of wmes.

$$\begin{array}{|l} TCI_{WME} : \mathbb{P}\ WME \rightarrow (\mathbb{P}\ Identifier \rightarrow \mathbb{P}\ Identifier) \\ \hline \forall W : \mathbb{P}\ WME;\ I : \mathbb{P}\ Identifier \bullet \\ \quad (TCI_{WME}(W))(I) = ((Connected_{WME}(W))^{*})(\!| I |\!) \end{array}$$

$OfId_{WME}$ extracts from a set of working memory elements, the ones that contain any element of a given set of identifiers in their id component.

$$\begin{array}{|l} OfId_{WME} : \mathbb{P}\ Identifier \times \mathbb{P}\ WME \rightarrow \mathbb{P}\ WME \\ \hline \forall I : \mathbb{P}\ Identifier;\ W : \mathbb{P}\ WME \bullet \\ \quad OfId_{WME}(I, W) = \{w : W \mid w.id \in I\} \end{array}$$

4.3 Temporary Memory Elements

Temporary memory is almost a disjoint union of working memory elements and preferences. Elements of temporary memory are of type *TME*, a disjoint union type of *Preference* and *WME*. Temporary memory is not exactly a disjoint union of working memory and preference memory. The one distinction is that the context acceptable preferences from preference memory are copied into working memory so that they can be matched by productions (see Chapter 5).

$$TME ::= PreferenceTME \langle\langle Preference \rangle\rangle \mid WMETME \langle\langle WME \rangle\rangle$$

A TME's components are defined to be the components of the underlying preference or working memory element.

$$\begin{array}{l}
TMEsComponents : TME \rightarrow \mathbb{P}\,\Sigma \\
\hline
\forall t : TME \bullet \\
\quad ((t \in \operatorname{ran} PreferenceTME \Rightarrow \\
\qquad TMEsComponents(t) = \\
\qquad\quad PreferencesComponents(PreferenceTME^{\sim}(t))) \land \\
\quad\ (t \in \operatorname{ran} WMETME \Rightarrow \\
\qquad TMEsComponents(t) = \\
\qquad\quad WMEsComponents(WMETME^{\sim}(t))))
\end{array}$$

$Connected_{TME}$ defines connectedness of identifiers through a single temporary memory element over a universe of temporary memory elements.

$$\begin{array}{l}
Connected_{TME} : \mathbb{P}\,TME \rightarrow (Identifier \leftrightarrow Identifier) \\
\hline
\forall T : \mathbb{P}\,TME;\ l, r : Identifier \bullet \\
\quad (l, r) \in Connected_{TME}(T) \Leftrightarrow \\
\qquad (l, r) \in \\
\qquad\quad Connected_{WME}(WMETME^{\sim}(\!| T \cap \operatorname{ran} WMETME |\!)) \cup \\
\qquad\quad Connected_{Preference}(PreferenceTME^{\sim}(\!| T \cap \operatorname{ran} PreferenceTME |\!))
\end{array}$$

TCI_{TME} extends the notion of connectedness to sets of identifiers over paths of temporary memory elements.

$$\begin{array}{l}
TCI_{TME} : \mathbb{P}\,TME \rightarrow (\mathbb{P}\,Identifier \rightarrow \mathbb{P}\,Identifier) \\
\hline
\forall T : \mathbb{P}\,TME;\ I : \mathbb{P}\,Identifier \bullet \\
\quad (TCI_{TME}(T))(I) = ((Connected_{TME}(T))^{*})(\!| I |\!)
\end{array}$$

OfId_{TME} extracts the subset of temporary memory elements that hold one of the given identifiers in their id slot.

$$\mathit{OfId}_{TME} : \mathbb{P}\ \mathit{Identifier} \times \mathbb{P}\ TME \rightarrow \mathbb{P}\ TME$$

$$\begin{array}{l}
\forall I : \mathbb{P}\ \mathit{Identifier};\ T : \mathbb{P}\ TME \bullet \\
\quad \mathit{OfId}_{TME}(I, T) = \\
\qquad \mathit{WMETME}(\!|\mathit{OfId}_{WME}(I, \mathit{WMETME}^{\sim}(\!|T \cap \operatorname{ran} \mathit{WMETME}|\!))|\!) \cup \\
\qquad \mathit{PreferenceTME}(\!|\mathit{OfId}_{Preference}(I, \mathit{PreferenceTME}^{\sim}(\!|T \cap \operatorname{ran} \mathit{PreferenceTME}|\!))|\!)
\end{array}$$

4.4 Makes

The process of recall from long term memory is modeled by the matching of productions. When productions match, their right hand sides are instantiated to find preferences to add to preference memory. The right hand side is a set of makes. The makes form a language that describes how productions create new preferences, given the variable bindings from the left hand side of the production match.

The data type schema, *UnaryMake* is instantiated to create a unary preference. Makes must contain a variable in the identifier slot, but can contain either a constant or a variable in the attribute and value fields.

$$\begin{array}{l}
\textit{UnaryMake} \\
\hline
\mathit{id} : \mathit{Variable} \\
\mathit{attribute}, \mathit{value} : \mathit{VariableOrConstant} \\
\mathit{preference} : \mathit{UnaryPreferenceSymbol} \\
\hline
\end{array}$$

A unary make's components is the set of its identifier, attribute and value.

$$\mathit{UnaryMakesComponents} : \mathit{UnaryMake} \rightarrow \mathbb{P}\ \Sigma$$

$$\begin{array}{l}
\forall u : \mathit{UnaryMake} \bullet \\
\quad \mathit{UnaryMakesComponents}(u) = \{u.\mathit{id}, u.\mathit{attribute}, u.\mathit{value}\}
\end{array}$$

A *BinaryMake* is instantiated to make a binary preference.

$$\begin{array}{l}
\textit{BinaryMake} \\
\hline
\mathit{id} : \mathit{Variable} \\
\mathit{attribute}, \mathit{value} : \mathit{VariableOrConstant} \\
\mathit{preference} : \mathit{BinaryPreferenceSymbol} \\
\mathit{referent} : \mathit{VariableOrConstant} \\
\hline
\end{array}$$

A binary make's components are the set of its identifier, attribute, value and referent.

$$\begin{array}{|l} BinaryMakesComponents : BinaryMake \rightarrow \mathbb{P}\,\Sigma \\ \hline \forall b : BinaryMake \bullet \\ \quad BinaryMakesComponents(b) = \{b.id, b.attribute, b.value, b.referent\} \end{array}$$

The free type *Make* is defined to join the phrases of the language of makes into a single phrase.

$$Make ::= UnaryM \langle\langle UnaryMake \rangle\rangle \mid BinaryM \langle\langle BinaryMake \rangle\rangle$$

A make's components are the components of the underlying unary or binary make.

$$\begin{array}{|l} MakesComponents : Make \rightarrow \mathbb{P}\,\Sigma \\ \hline \forall m : Make \bullet \\ \quad ((m \in \operatorname{ran} UnaryM \Rightarrow \\ \quad\quad MakesComponents(m) = \\ \quad\quad\quad UnaryMakesComponents(UnaryM^{\sim}(m))) \wedge \\ \quad (m \in \operatorname{ran} BinaryM \Rightarrow \\ \quad\quad MakesComponents(m) = \\ \quad\quad\quad BinaryMakesComponents(BinaryM^{\sim}(m)))) \end{array}$$

makeUnaryMake and *makeBinaryMake* instantiate copies of the unary or binary make schema, and wrap them in the make free type.

$$
\begin{array}{|l}
\mathit{makeUnaryMake} : \\
\quad \mathit{Identifier} \times \mathit{VariableOrConstant} \times \mathit{VariableOrConstant} \times \\
\qquad \mathit{UnaryPreferenceSymbol} \rightarrow \mathit{Make} \\
\mathit{makeBinaryMake} : \\
\quad \mathit{Identifier} \times \mathit{VariableOrConstant} \times \mathit{VariableOrConstant} \times \\
\qquad \mathit{BinaryPreferenceSymbol} \times \mathit{VariableOrConstant} \rightarrow \mathit{Make} \\
\hline
\forall i : \mathit{Identifier};\ a, v : \mathit{VariableOrConstant};\ p : \mathit{UnaryPreferenceSymbol} \bullet \\
\quad \mathit{makeUnaryMake}(i, a, v, p) = \\
\qquad \mathit{UnaryM}((\mu\ \mathit{UnaryPreference} \mid \\
\qquad\qquad \mathit{id} = i \land \mathit{attribute} = a \land \mathit{value} = v \land \mathit{preference} = p \bullet \\
\qquad\quad \theta\, \mathit{UnaryMake})) \\
\\
\forall i : \mathit{Identifier};\ a, v, r : \mathit{VariableOrConstant};\ p : \mathit{BinaryPreferenceSymbol} \bullet \\
\quad \mathit{makeBinaryMake}(i, a, v, p, r) = \\
\qquad \mathit{BinaryM}((\mu\ \mathit{BinaryPreference} \mid \\
\qquad\qquad \mathit{id} = i \land \mathit{attribute} = a \land \mathit{value} = v \land \\
\qquad\qquad \mathit{preference} = p \land \mathit{referent} = r \bullet \\
\qquad\quad \theta\, \mathit{BinaryMake}))
\end{array}
$$

$Connected_{Make}$ defines connectedness for makes. It is parallel to the connectedness for preferences, with the exception that makes hold constants or variables in their components instead of constants or identifiers, but connectedness only flows through variables.

$$
\begin{array}{|l}
\mathit{Connected}_{\mathit{Make}} : \mathbb{P}\ \mathit{Make} \rightarrow (\mathit{Variable} \leftrightarrow \mathit{Variable}) \\
\hline
\forall M : \mathbb{P}\ \mathit{Make};\ l, r : \mathit{Variable} \bullet \\
\quad (l, r) \in (\mathit{Connected}_{\mathit{Make}}(M)) \Leftrightarrow \\
\qquad (\exists\, m : M \bullet \\
\qquad\quad ((m \in \operatorname{ran} \mathit{UnaryM} \Rightarrow \\
\qquad\qquad (\exists\, u : \mathit{UnaryMake} \mid u = \mathit{UnaryM}^{\sim}(m) \bullet \\
\qquad\qquad\quad u.\mathit{id} = l \land u.\mathit{value} = r)) \land \\
\qquad\quad (m \in \operatorname{ran} \mathit{BinaryM} \Rightarrow \\
\qquad\qquad (\exists\, b : \mathit{BinaryMake} \mid b = \mathit{BinaryM}^{\sim}(m) \bullet \\
\qquad\qquad\quad b.\mathit{id} = l \land (b.\mathit{value} = r \lor b.\mathit{referent} = r)))))
\end{array}
$$

TCI_{Make} extends the definition of connectedness to cover sets of variables and paths of makes.

$$
\begin{array}{l}
TCI_{Make} : \mathbb{P}\ Make \rightarrow (\mathbb{P}\ Variable \rightarrow \mathbb{P}\ Variable) \\
\hline
\forall M : \mathbb{P}\ Make;\ V : \mathbb{P}\ Variable \bullet \\
\quad (TCI_{Make}(M))(V) = ((Connected_{Make}(M))^{*})(\!| V |\!)
\end{array}
$$

$OfId_{Make}$ extracts a subset of a set of makes that holds any element from a set of variables in their identifier slot.

$$
\begin{array}{l}
OfId_{Make} : \mathbb{P}\ Variable \times \mathbb{P}\ Make \rightarrow \mathbb{P}\ Make \\
\hline
\forall V : \mathbb{P}\ Variable;\ M : \mathbb{P}\ Make \bullet \\
\quad OfId_{Make}(V, M) = \\
\quad\quad \{m : M \mid \\
\quad\quad\quad (m \in \operatorname{ran} UnaryM \Rightarrow (UnaryM^{\sim}(m)).id \in V) \vee \\
\quad\quad\quad (m \in \operatorname{ran} BinaryM \Rightarrow (BinaryM^{\sim}(m)).id \in V)\}
\end{array}
$$

4.5 Tests

Productions require a language to describe how to match against the contents of working memory. The terminal phrases of this language are called *Tests* and match against the symbols found in the fields of working memory elements.

Test is defined as a free type, both to construct a disjoint union and to allow tests to recursively contain tests ([Spi89] 81-83). Three of the test types are constant, and five are constructed from arguments, one of which takes a set of tests as arguments.

$$
\begin{array}{rl}
Test ::= & BlankTest \\
\mid & YesTest \\
\mid & NoTest \\
\mid & EqualityTest\langle\!\langle \Sigma \rangle\!\rangle \\
\mid & NotTest\langle\!\langle \Sigma \rangle\!\rangle \\
\mid & SameTypeTest\langle\!\langle \Sigma \rangle\!\rangle \\
\mid & DisjunctiveTest\langle\!\langle \mathbb{P}\ Constant \rangle\!\rangle \\
\mid & ConjunctiveTest\langle\!\langle \mathbb{P}\ Test \rangle\!\rangle
\end{array}
$$

As a production matches, it constructs a binding of variables to symbols, much like an environment in a programming language. When a test matches against a symbol, if the test is constructed using a variable it will match using the binding of the variable instead of the variable itself.

There are eight types of tests.

1. BlankTest — matches any symbol.
2. YesTest — matches against a value of type *YesOrNo* that is a *Yes*.
3. NoTest — matches against a value of type *YesOrNo* that is a *No*.
4. EqualityTest — matches against a symbol if it is the same symbol used to create the equality test, or the binding of the variable used to create the test.
5. NotTest — matches against a symbol, only if is not the same symbol used to create the NotTest, or not the same symbol as the binding of the variable used to create the test.
6. SameTypeTest — matches against a symbol only if the symbol and the constructing argument of the *SameTypeTest* (or its binding) are both in *Identifier* or both in *Constant*.
7. DisjunctiveTest — matches a symbol only if it is in the set of constants used to create the disjunctive test.
8. ConjunctiveTest — matches a symbol only if the symbol matches all of the set of tests that were used to create the conjunction.

Although Soar could support recursive conjunctive tests they provide no extra matching power over a single layer of conjunctions. The predicate below constrains all conjunctive tests to be constructed only from equality tests, not tests, same type tests and disjunctions.

$$\begin{array}{l}
\forall\, ct : \operatorname{ran} ConjunctiveTest \bullet \\
\quad ConjunctiveTest^{\sim}(ct) \subseteq \\
\qquad (\operatorname{ran} EqualityTest \cup \operatorname{ran} NotTest \cup \\
\qquad\ \operatorname{ran} SameTypeTest \cup \operatorname{ran} DisjunctiveTest)
\end{array}$$

The components of tests are collected from the arguments used to construct the test.

$$
\begin{array}{|l}
\mathit{TestsComponents} : \mathit{Test} \rightarrow \mathbb{P}\,\Sigma \\
\hline
\forall t : \mathit{Test} \bullet \\
\quad ((t \in \{\mathit{BlankTest}, \mathit{YesTest}, \mathit{NoTest}\} \Rightarrow \\
\qquad \mathit{TestsComponents}(t) = \emptyset) \land \\
\quad (t \in \operatorname{ran} \mathit{EqualityTest} \Rightarrow \\
\qquad \mathit{TestsComponents}(t) = \{\mathit{EqualityTest}^{\sim}(t)\}) \land \\
\quad (t \in \operatorname{ran} \mathit{NotTest} \Rightarrow \\
\qquad \mathit{TestsComponents}(t) = \{\mathit{NotTest}^{\sim}(t)\}) \land \\
\quad (t \in \operatorname{ran} \mathit{SameTypeTest} \Rightarrow \\
\qquad \mathit{TestsComponents}(t) = \{\mathit{SameTypeTest}^{\sim}(t)\}) \land \\
\quad (t \in \operatorname{ran} \mathit{DisjunctiveTest} \Rightarrow \\
\qquad \mathit{TestsComponents}(t) = \mathit{DisjunctiveTest}^{\sim}(t)) \land \\
\quad (t \in \operatorname{ran} \mathit{ConjunctiveTest} \Rightarrow \\
\qquad \mathit{TestsComponents}(t) = \bigcup(\mathit{TestsComponents}(\!| \mathit{ConjunctiveTest}^{\sim}(t) |\!))))
\end{array}
$$

Soar's conditions have a notion of a positive versus a negative test. A positive test is one that checks for equality with a constant or a variable's binding, and a negative test is one that does not. The *TestsPositiveComponents* mapping is defined to extract the set of constants and variables that a test positively checks.

$$
\begin{array}{|l}
\mathit{TestsPositiveComponents} : \mathit{Test} \rightarrow \mathbb{P}\,\mathit{Symbol} \\
\hline
\forall t : \mathit{Test} \bullet \\
\quad ((t \in \{\mathit{BlankTest}, \mathit{YesTest}, \mathit{NoTest}\} \cup \\
\qquad \operatorname{ran} \mathit{SameTypeTest} \cup \operatorname{ran} \mathit{NotTest} \Rightarrow \\
\qquad\quad \mathit{TestsPositiveComponents}(t) = \emptyset) \land \\
\quad (t \in \operatorname{ran} \mathit{EqualityTest} \Rightarrow \\
\qquad \mathit{TestsPositiveComponents}(t) = \{\mathit{EqualityTest}^{\sim}(t)\}) \land \\
\quad (t \in \operatorname{ran} \mathit{DisjunctiveTest} \Rightarrow \\
\qquad \mathit{TestsPositiveComponents}(t) = \mathit{DisjunctiveTest}^{\sim}(t)) \land \\
\quad (t \in \operatorname{ran} \mathit{ConjunctiveTest} \Rightarrow \\
\qquad \mathit{TestsPositiveComponents}(t) = \\
\qquad\quad \bigcup(\mathit{TestsPositiveComponents}(\!| \mathit{ConjunctiveTest}^{\sim}(t) |\!))))
\end{array}
$$

While describing operations on tests, it is often necessary to describe one action for conjunctive tests and another action for all other types of tests. The abbreviation definition ([Spi89] 52) below defines a new type set, *SimpleTest*, that simplifies checking whether a test is not conjunctive.

$SimpleTest == (Test \setminus \operatorname{ran} ConjunctiveTest)$

4.6 WME Tests

Tests are composed in structures, called a *WMETest*, that tests against a working memory element. A WMETest matches a working memory element if and only if all of the working memory element test's tests match the corresponding fields of the working memory element.

WMETest

$id, attribute : Test \setminus \{BlankTest, YesTest, NoTest\}$
$value : Test \setminus \{YesTest, NoTest\}$
$context\_acceptable\_preference : \{YesTest, NoTest\}$

makeWMETest instantiates a working memory element test given its four component tests.

$$\begin{aligned}
&makeWMETest : Test \setminus \{BlankTest, YesTest, NoTest\} \times \\
&\quad Test \setminus \{YesTest, NoTest\} \times \\
&\quad Test \setminus \{YesTest, NoTest\} \times \\
&\quad \{YesTest, NoTest\} \times \{YesTest, NoTest\} \rightarrow WMETest
\end{aligned}$$

$$\begin{aligned}
&\forall\, id : Test \setminus \{BlankTest, YesTest, NoTest\}; \\
&\quad attribute, value : Test \setminus \{YesTest, NoTest\}; \\
&\quad context\_acceptable\_preference : \{YesTest, NoTest\} \bullet \\
&\quad makeWMETest(id, attribute, value, context\_acceptable\_preference) = \\
&\quad (\mu\, id : \{id\};\ attribute : \{attribute\};\ value : \{value\}; \\
&\quad\quad context\_acceptable\_preference : \{context\_acceptable\_preference\} \bullet \\
&\quad\quad\quad \theta WMETest)
\end{aligned}$$

The components of a working memory element test are the components of its id, attribute and value fields.

$WMETestsComponents : WMETest \rightarrow \mathbb{P}\ Symbol$

$$\begin{aligned}
&\forall\, wt : WMETest \bullet \\
&\quad WMETestsComponents(wt) = \\
&\quad\quad \bigcup(TestsComponents(\{wt.id, wt.attribute, wt.value\}))
\end{aligned}$$

WMETestsPositiveComponents extends the concept of a positive component to the level of a working memory element test.

$$
\begin{array}{l}
\mathit{WMETestsPositiveComponents} : \mathit{WMETest} \rightarrow \mathbb{P}\ \mathit{Symbol} \\
\hline
\forall\, wt : \mathit{WMETest} \bullet \\
\quad \mathit{WMETestsPositiveComponents}(wt) = \\
\qquad \bigcup(\mathit{TestsPositiveComponents}\,(\!|\{wt.id,\ wt.attribute,\ wt.value\}|\!))
\end{array}
$$

Two working memory element tests are connected if either of their identifier tests share positive components, or the first test's value test shares components with the second one's id test.

$$
\begin{array}{l}
\_\ \mathit{Connected}_{\mathit{WMETest}}\ \_ : \mathit{WMETest} \leftrightarrow \mathit{WMETest} \\
\hline
\forall\, wt_1, wt_2 : \mathit{WMETest} \bullet \\
\quad wt_1\ \mathit{Connected}_{\mathit{WMETest}}\ wt_2 \Leftrightarrow \\
\qquad ((\mathit{TestsPositiveComponents}(wt_1.id) \cap \\
\qquad\quad \mathit{TestsPositiveComponents}(wt_2.id)) \neq \emptyset \lor \\
\qquad (\mathit{TestsPositiveComponents}(wt_1.value) \cap \\
\qquad\quad \mathit{TestsPositiveComponents}(wt_2.id)) \neq \emptyset)
\end{array}
$$

4.7 Conditions

Soar's productions check working memory for the existence of elements matching working memory element tests, but they must also check if working memory does not contain matching patterns of elements. Working memory element tests are bundled into structures, called *Conditions*, that match on the existence and non-existence of patterns of elements.

PositiveConditions match when there exists a working memory element that their *WMETest* matches. *NegativeCondition* match the non-existence of a element that matches their *WMETest*. *NegativeConjunctiveConditions* match when all of their conditions fail to simultaneously match working memory.

$$
\begin{array}{rl}
\mathit{Condition} ::= & \mathit{PositiveCondition}\langle\!\langle \mathit{WMETest} \rangle\!\rangle \\
| & \mathit{NegativeCondition}\langle\!\langle \mathit{WMETest} \rangle\!\rangle \\
| & \mathit{NegativeConjunctiveCondition}\langle\!\langle \mathbb{P}_1\ \mathit{Condition} \rangle\!\rangle
\end{array}
$$

Conjunctive negations are restricted to contain at least one positive condition.

$$\begin{array}{l}
\forall c : \operatorname{ran} \mathit{NegativeConjunctiveCondition} \bullet \\
\quad \exists cs : \mathbb{P}\, \mathit{Condition} \mid cs = \mathit{NegativeConjunctiveCondition}^{\sim}(c) \bullet \\
\qquad \operatorname{ran} \mathit{PositiveCondition} \cap cs \neq \emptyset
\end{array}$$

While specifying operations on conditions cased by type, it is convenient to have a sub-type of conditions, *SimpleCondition*, that excludes the negative conjunctive conditions.

$$\mathit{SimpleCondition} == \operatorname{ran} \mathit{PositiveCondition} \cup \operatorname{ran} \mathit{NegativeCondition}$$

ConditionsComponents extracts the components of a condition from its working memory element tests or recursively from its embedded conditions.

$$\begin{array}{l}
\mathit{ConditionsComponents} : \mathit{Condition} \rightarrow \mathbb{P}\, \mathit{Symbol} \\
\hline
\forall c : \mathit{Condition} \bullet \\
\quad ((c \in \operatorname{ran} \mathit{PositiveCondition} \Rightarrow \\
\qquad \mathit{ConditionsComponents}(c) = \\
\qquad\quad \mathit{WMETestsComponents}(\mathit{PositiveCondition}^{\sim}(c))) \wedge \\
\quad (c \in \operatorname{ran} \mathit{NegativeCondition} \Rightarrow \\
\qquad \mathit{ConditionsComponents}(c) = \\
\qquad\quad \mathit{WMETestsComponents}(\mathit{NegativeCondition}^{\sim}(c))) \wedge \\
\quad (c \in \operatorname{ran} \mathit{NegativeConjunctiveCondition} \Rightarrow \\
\qquad \mathit{ConditionsComponents}(c) = \\
\qquad\quad \bigcup(\mathit{ConditionsComponents}(\!|\mathit{NegativeConjunctiveCondition}^{\sim}(c)|\!))))
\end{array}$$

The positive components of a condition are defined to be all of the positive components of its working memory element test or embedded conditions.

$$\begin{array}{l}
\mathit{ConditionsPositiveComponents} : \mathit{Condition} \rightarrow \mathbb{P}\, \mathit{Symbol} \\
\hline
\forall c : \mathit{Condition} \bullet \\
\quad ((c \in \operatorname{ran} \mathit{PositiveCondition} \Rightarrow \\
\qquad \mathit{ConditionsPositiveComponents}(c) = \\
\qquad\quad \mathit{WMETestsPositiveComponents}(\mathit{PositiveCondition}^{\sim}(c))) \wedge \\
\quad (c \in \operatorname{ran} \mathit{NegativeCondition} \Rightarrow \\
\qquad \mathit{ConditionsPositiveComponents}(c) = \\
\qquad\quad \mathit{WMETestsPositiveComponents}(\mathit{NegativeCondition}^{\sim}(c))) \wedge \\
\quad (c \in \operatorname{ran} \mathit{NegativeConjunctiveCondition} \Rightarrow \\
\qquad \mathit{ConditionsPositiveComponents}(c) = \\
\qquad\quad \bigcup(\mathit{ConditionsPositiveComponents}(\!|\mathit{NegativeConjunctiveCondition}^{\sim}(c)|\!))))
\end{array}$$

Soar constrains the conditions of the left hand side of the production to be connected, and chunking's inner loops calculate the transitive closures of conditions. Unfortunately, negated conditions and conjunctive negations do not allow the definition of connectedness of conditions to be parallel with those of preferences, working memory elements, temporary memory elements and makes.

$Connected_{Condition}$ is defined to connect sets of conditions to sets of conditions, instead of connecting pairs of identifiers over a universe of conditions. This allows the definition to support Soar's use of multiple starting conditions to connect into a conjunctive negation.

Two sets of conditions, A and B, are connected if and only if B holds a positive condition, b, whose *WMETest* is connected to the *WMETest* of any positive condition in A, and A union the set of b is connected to B's other conditions. Similarly, a negated condition, b in B, is connected to A if there is a positive condition in A whose *WMETest* connects to b, and A connects to the remaining elements of B. Notice that only positive conditions in A may connect to a elements of B.

A set of conditions, A, is connected to a set of conditions, B, containing a conjunctive negation, b if and only if A connects to all of the sub-conditions of b and A union the set of b connects to B's remaining conditions. This connects any conjunctive negation whose sub-conditions are all connected to A, either directly or through a path in the sub-conditions.

$$\_\,Connected_{Condition}\,\_ : \mathbb{P}\ Condition \leftrightarrow \mathbb{P}\ Condition$$

$$\begin{array}{l}
\forall A : \mathbb{P}\ Condition \bullet A\ Connected_{Condition}\ \varnothing \\
\forall A, B : \mathbb{P}\ Condition \bullet \\
\quad A\ Connected_{Condition}\ B \Leftrightarrow \\
\quad\quad (\exists b : B \bullet \\
\quad\quad\quad (b \in \mathrm{ran}\ PositiveCondition \Rightarrow \\
\quad\quad\quad\quad (\exists a : A \bullet \\
\quad\quad\quad\quad\quad PositiveCondition^{\sim}(a)\ Connected_{WMETest}\ PositiveCondition^{\sim}(b))) \wedge \\
\quad\quad\quad (b \in \mathrm{ran}\ NegativeCondition \Rightarrow \\
\quad\quad\quad\quad (\exists a : A \bullet \\
\quad\quad\quad\quad\quad PositiveCondition^{\sim}(a)\ Connected_{WMETest}\ NegativeCondition^{\sim}(b))) \wedge \\
\quad\quad\quad (b \in \mathrm{ran}\ NegativeConjunctiveCondition \Rightarrow \\
\quad\quad\quad\quad (A\ Connected_{Condition}\ NegativeConjunctiveCondition^{\sim}(b))) \wedge \\
\quad\quad\quad A \cup \{b\}\ Connected_{Condition}\ B \setminus \{b\})
\end{array}$$

$TCl_{Condition}$ transitively closes a set of conditions S over a universe U. The definition constructs the closure using the maximal set under the set inclusion ordering. The image of a form of the reflexive transitive closure could be used, but it would be cumbersome due to $Connected_{Condition}$'s different signature.

$$
\begin{array}{l}
TCl_{Condition} : \mathbb{P}\ Condition \times \mathbb{P}\ Condition \rightarrow \mathbb{P}\ Condition \\
\hline
\forall S, U : \mathbb{P}\ Condition \bullet \\
\quad \exists C : \mathbb{P}\ U \mid \\
\quad\quad (S\ Connected_{Condition}\ C) \wedge \\
\quad\quad (\forall B : \mathbb{P}\ U \bullet S\ Connected_{Condition}\ B \Rightarrow B \subseteq C) \bullet \\
\quad TCl_{Condition}(S, U) = C
\end{array}
$$

$OfId_{Condition}$ extracts the positive and negative conditions of a set that match any constant in their identifier. It does not extract the negative conjunctive conditions that hold the set of symbols in their identifier because this requires a full recursive definition of connectedness. Where this is required, $OfId_{Condition}$ is used in conjunction with $Connected_{Condition}$ so that it can connect into the conjunctive negations.

$$
\begin{array}{l}
OfId_{Condition} : \mathbb{P}\ \Sigma \times \mathbb{P}\ Condition \rightarrow \mathbb{P}\ Condition \\
\hline
\forall I : \mathbb{P}\ \Sigma;\ C : \mathbb{P}\ Condition \bullet \\
\quad OfId_{Condition}(I, C) = \\
\quad\quad \{c : C \mid \exists i : I \bullet \\
\quad\quad\quad (c \in \operatorname{ran} PositiveCondition \Rightarrow \\
\quad\quad\quad\quad i \in TestsPositiveComponents((PositiveCondition^{\sim}(c)).id)) \wedge \\
\quad\quad\quad (c \in \operatorname{ran} NegativeCondition \Rightarrow \\
\quad\quad\quad\quad i \in TestsPositiveComponents((NegativeCondition^{\sim}(c)).id))\}
\end{array}
$$

4.8 SP

Productions are Soar's model of long term memories. They match against the contents of working memory, instantiate and fire to augment support memory with preferences.

Soar's definition of match requires that every variable that is tested against (with NotTest, or SameTypeTest) must be bound in some condition (with an EqualityTest). *TestedVariablesBound* is a predicate that checks that the variables a condition tests against are all in the set of bound variables

$$
\begin{array}{l}
\mathit{TestedVariablesBound}\ \_ : \mathbb{P}(\mathit{Condition} \times \mathbb{P}\ \mathit{Variable}) \\
\hline
\forall\, \mathit{bound} : \mathbb{P}\ \mathit{Variable};\ c : \mathit{Condition} \bullet \\
\quad \mathit{TestedVariablesBound}\ (c, \mathit{bound}) \Leftrightarrow \\
\quad\quad (c \in \operatorname{ran} \mathit{PositiveCondition} \Rightarrow \\
\quad\quad\quad (\mathit{ConditionsComponents}(c) \cap \mathit{Variable}) \subseteq \mathit{bound}) \wedge \\
\quad\quad (c \in \operatorname{ran} \mathit{NegativeCondition} \Rightarrow \\
\quad\quad\quad (\mathit{ConditionsComponents}(c) \cap \mathit{Variable}) \subseteq \\
\quad\quad\quad\quad (\mathit{bound} \cup (\mathit{ConditionsPositiveComponents}(c) \cap \mathit{Variable}))) \wedge \\
\quad\quad (c \in \operatorname{ran} \mathit{NegativeConjunctiveCondition} \Rightarrow \\
\quad\quad\quad (\exists\, cs : \{\mathit{NegativeConjunctiveCondition}^{\sim}(c)\} \bullet \\
\quad\quad\quad (\exists\, bv : \mathbb{P}\ \mathit{Variable} \mid \\
\quad\quad\quad\quad bv = \mathit{bound} \cup \\
\quad\quad\quad\quad\quad (\bigcup(\mathit{ConditionsPositiveComponents}(\!| cs \cap \operatorname{ran} \mathit{PositiveCondition} |\!)) \\
\quad\quad\quad\quad\quad\quad \cap\ \mathit{Variable}) \bullet \\
\quad\quad\quad\quad (\forall\, c : cs \bullet \mathit{TestedVariablesBound}\ (c, bv)))))
\end{array}
$$

Soar's productions, SPs, have a name, a set of left hand side conditions, lhs, that describe how they match against working memory and a right hand side set of actions, rhs. The lhs must contain at least one positive condition, or the rhs would have no variables bound to use in make actions. Each condition may only test against variables that are bound with a equality test somewhere on the left hand side.

$$
\begin{array}{|l}
\textit{SP} \\
\hline
\textit{name} : \textit{Symbol} \\
\textit{lhs} : \mathbb{P}_1\ \textit{Condition} \\
\textit{rhs} : \mathbb{P}\ \textit{Make} \\
\textit{roots} : \mathbb{P}\ \textit{Variable} \\
\hline
\exists\, c : \textit{lhs} \bullet c \in \operatorname{ran} \textit{PositiveCondition} \\
\forall\, c : \textit{lhs} \bullet \textit{ConditionsComponents}(c) \cap \textit{Identifier} = \emptyset \\
\textit{roots} = \\
\quad \{ v : \bigcup(\textit{ConditionsPositiveComponents}\,(\!|(\textit{lhs} \cap \operatorname{ran} \textit{PositiveCondition})|\!)) \\
\qquad \neg\,(\exists\, c : \textit{lhs} \cap \operatorname{ran} \textit{PositiveCondition} \bullet \\
\qquad\quad v \in \textit{TestsPositiveComponents}((\textit{PositiveCondition}^{\sim}(c)).\textit{value}))\} \\
\textit{lhs} = \textit{TCI}_{\textit{Condition}}(\textit{OfId}_{\textit{Condition}}(\textit{roots}, \textit{lhs}), \textit{lhs}) \\
\exists\, \textit{boundvariables} : \mathbb{P}\ \textit{Variable} \mid \\
\quad \textit{boundvariables} = \\
\qquad (\bigcup(\textit{ConditionsPositiveComponents}\,(\!|\textit{lhs} \cap \operatorname{ran} \textit{PositiveCondition}|\!)) \\
\qquad\qquad \cap\ \textit{Variable}) \bullet \\
\qquad (\forall\, c : \textit{lhs} \bullet \textit{TestedVariablesBound}\ (c, \textit{boundvariables}) \wedge \\
\qquad \textit{rhs} = \textit{OfId}_{\textit{Make}}(\textit{TCI}_{\textit{Make}}(\textit{rhs})(\textit{boundvariables}), \textit{rhs})) \\
\hline
\end{array}
$$

Conditions are stored in two places in Soar: in productions they describe when and how productions match, and in chunking they record which patterns their productions matched against. Chunking's conditions may have matched against identifiers and so they are instantiated with identifiers to record the match. However, productions are constrained not to use identifiers in their match, otherwise they would be terribly specific to the set of identifiers in use (except for internal chunks).

A root of a production is an identifier that is not positively tested in any value of the positive conditions. Soar constrains these variables to match against goal or impasse identifiers. All of the conditions of the lhs must be in the transitive closure of the roots; this ensures that productions only match working memory elements which are in the transitive closure of the context stack or an impasse.

The right hand side is also constrained to be transitively closed over the bound variables of the left hand side. This ensures that the instantiations only make new preferences that are accessible from a goal or impasse.

makeSP is used to construct a new production data schema. It takes only three arguments, the name, the left hand side and the right hand side, as the roots are a derived component of the left hand side.

$$
\begin{array}{l}
makeSP : Symbol \times \mathbb{P}_1\ Condition \times \mathbb{P}\ Make \rightarrow SP \\
\hline
\forall\, name : Symbol;\ lhs : \mathbb{P}_1\ Condition;\ rhs : \mathbb{P}\ Make \bullet \\
\quad \exists_1\, roots : \mathbb{P}\ Variable \mid SP \bullet \\
\quad\quad makeSP(name, lhs, rhs) = \theta SP
\end{array}
$$

The components of productions are all of the components of its left hand side and the components of its right hand side.

$$
\begin{array}{l}
SpsComponents : SP \rightarrow \mathbb{P}\ \Sigma \\
\hline
\forall\, sp : SP \bullet \\
\quad SpsComponents(sp) = \\
\quad\quad \bigcup(ConditionsComponents(\!| sp.lhs |\!) \cup MakesComponents(\!| sp.rhs |\!))
\end{array}
$$

Chapter 5

Recognition Memory

Recognition memory is where Soar holds, processes and acquires long term memories. Long term memories are stored as productions that match against short term memories in working memory. Recognition memory matches the productions against working memory, stores the matches, instantiates matches to produce new preferences, and calculates support for preferences and chunks.

The chapter is divided into nine sections:

- Section 5.1 Bindings — introduces bindings that map variables to values
- Section 5.2 Matching — defines how productions match against working memory
- Section 5.3 Instantiation — specifies how matches are instantiated
- Section 5.4 Goal Operations — defines some operations on Decide's goal stack for use by support
- Section 5.5 Chunking — defines chunking, Soar's learning mechanism
- Section 5.6 Slots — defines the concepts of a slot.
- Section 5.7 Recognition Memory's State — constructs the state of recognition memory
- Section 5.8 Support — specifies how instantiations support preferences and
- Section 5.9 Preference Phase Operations — defines the state machine for recognition memory's control loop, called preference phase.

5.1 Bindings

Soar's productions match against working memory using a binding of the production's variables to symbols. Bindings are similar to environments in programming languages. In Soar, bindings are modeled as partial functions from variables to symbols.

$$\mathit{Binding} == \mathit{Variable} \nrightarrow \mathit{Symbol}$$

When productions match, they check that there is no possible way to instantiate their negated conditions to allow them to match working memory. The negative conditions must not be able to add bindings for their local variables that allow them to match working memory elements. To formalize the match of negative and negative conjunctive conditions, we define the concepts of two bindings being consistent, a binding covering a set of variables, and one binding being a consistent extension of another.

Two bindings are consistent if they both bind every variable that they share to the same symbol. A binding covers a set of variables if it gives a binding to every variable in the set. A binding, A, is a consistent extension of another binding, B, if it is consistent with it and A possibly binds variables that B does not.

$$\begin{array}{l}
\_\ \mathit{Consistent}\ \_, \_\ \mathit{ConsistentExtension}\ \_ : \mathit{Binding} \leftrightarrow \mathit{Binding} \\
\_\ \mathit{Covers}\ \_ : \mathit{Binding} \leftrightarrow \mathbb{P}\,\Sigma \\
\hline
\forall\, a, b : \mathit{Binding} \bullet \\
\quad (a\ \mathit{Consistent}\ b) \Leftrightarrow \\
\quad\quad (\forall\, v : \mathrm{dom}\, a \cap \mathrm{dom}\, b \bullet a(v) = b(v)) \\
\\
\forall\, b : \mathit{Binding} \bullet \forall\, r : \mathbb{P}\ \mathit{Variable} \bullet \\
\quad b\ \mathit{Covers}\ r \Leftrightarrow r \subseteq \mathrm{dom}\, b \\
\\
\forall\, a, b : \mathit{Binding} \bullet \\
\quad a\ \mathit{ConsistentExtension}\ b \Leftrightarrow \\
\quad\quad (a\ \mathit{Consistent}\ b \wedge (\mathrm{dom}\, b \subseteq \mathrm{dom}\, a))
\end{array}$$

5.2 Matching

The core result of matching a production against working memory is a correspondence between the positive conditions of the production and the working memory elements they match. In this section, we define how a test matches a symbol, and how a working memory element test matches a working memory element. We use this to define the concept of a match to a production, and then define when a match is consistent with the contents of working memory.

5.2.1 Tests Matching Symbols

A test can match a symbol, under a binding, in one of six ways.

1. The test can be the blank test, which matches any symbol.

2. Equality tests only match a symbol, A, if their argument is A, or a variable bound to A. The $\oplus$ notation is Z's way of function overriding ([Spi89] 108). $(a \oplus b)(x)$ is $b(x)$ if $x \in \mathrm{dom}\, b$ or $a(x)$ if $x \in \mathrm{dom}\, a$ and undefined otherwise. The $(\mathrm{id}\,\Sigma)$ is the identity function on Σ ([Spi89] 97). The overriding of $(\mathrm{id}\,\Sigma)$ with the binding, b, produces a function that returns the binding of the equality test's variable, or the equality test's constant.

3. Not tests match a symbol, A, only if their symbol or variable's binding is not equal to A.

4. Same type tests match a symbol, A, against their symbol or variable's binding, B, only if A and B are both identifiers or both constants.

5. A disjunctive test matches only constants that are elements of the disjunction's argument set.

6. Conjunctive tests match a symbol only if the symbol matches all of the conjunction's sub-tests.

$$
\begin{array}{|l}
\mathit{Matches}^{\mathrm{Test}}_{\mathit{Symbol}}\ \_ : \mathbb{P}(\mathit{Binding} \times \mathit{Test} \times \mathit{Symbol}) \\
\hline
\forall\, b : \mathit{Binding};\ t : \mathit{Test};\ v : \mathit{Symbol} \bullet \\
\quad ((t = \mathit{BlankTest}) \lor \\
\quad\ (t \in \mathrm{ran}\, \mathit{EqualityTest} \land v = ((\mathrm{id}\,\Sigma) \oplus b)(\mathit{EqualityTest}^{\sim}(t))) \lor \\
\quad\ (t \in \mathrm{ran}\, \mathit{NotTest} \land v \neq ((\mathrm{id}\,\Sigma) \oplus b)(\mathit{NotTest}^{\sim}(t))) \lor \\
\quad\ (t \in \mathrm{ran}\, \mathit{SameTypeTest} \land \\
\quad\quad (\exists\, v2 : \{((\mathrm{id}\,\mathit{Symbol}) \oplus b)(\mathit{SameTypeTest}^{\sim}(t))\} \bullet \\
\quad\quad\quad ((v \in \mathit{Identifier} \land v2 \in \mathit{Identifier}) \lor \\
\quad\quad\quad\ (v \in \mathit{Constant} \land v2 \in \mathit{Constant})))) \land \\
\quad\ (t \in \mathrm{ran}\, \mathit{DisjunctiveTest} \land v \in \mathit{DisjunctiveTest}^{\sim}(t)) \lor \\
\quad\ (t \in \mathrm{ran}\, \mathit{ConjunctiveTest} \land \\
\quad\quad (\forall\, \mathit{subt} : \mathit{ConjunctiveTest}^{\sim}(t) \bullet \mathit{Matches}^{\mathrm{Test}}_{\mathit{Symbol}}\ (b, \mathit{subt}, v))))
\end{array}
$$

A working memory element test matches a working memory element if its id, attribute, value and context acceptable preference fields match the element's test's id, attribute, value and context acceptable preference test.

$$Matches_{WME}^{WMETest}\ \_ : \mathbb{P}(Binding \times WMETest \times WME)$$

$$
\begin{array}{l}
\forall\, b : Binding;\ wt : WMETest;\ w : WME \bullet \\
\quad Matches_{WME}^{WMETest}\ (b, wt, w) \Leftrightarrow \\
\quad\quad (Matches_{Symbol}^{Test}\ (b, wt.id, w.id) \wedge \\
\quad\quad\ Matches_{Symbol}^{Test}\ (b, wt.attribute, w.attribute) \wedge \\
\quad\quad\ Matches_{Symbol}^{Test}\ (b, wt.value, w.value) \wedge \\
\quad\quad\ (wt.context\_acceptable\_preference = YesTest \Rightarrow \\
\quad\quad\quad w.context\_acceptable\_preference = Yes) \wedge \\
\quad\quad\ (wt.context\_acceptable\_preference = NoTest \Rightarrow \\
\quad\quad\quad w.context\_acceptable\_preference = No))
\end{array}
$$

5.2.2 Matches

The full result of a production matching is recorded in a match schema. The match contains: the production that matched, the correspondence between its positive conditions and the working memory elements it matched, called a matching, the binding that allows this correspondence, and the set of matched working memory elements.

The matching is actually a correspondence between the positive conditions and working memory element, number pairs. As each working memory element could be added to working memory, later removed and the re-added, we identify each instance of a working memory element uniquely using a natural number tag.

$$
\begin{array}{l}
\underline{\quad Match \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
\mathit{production} : SP \\
\mathit{matching} : \operatorname{ran} \mathit{PositiveCondition} \rightarrowtail (\mathit{WME} \times \mathbb{N}) \\
\mathit{binding} : \mathit{Binding} \\
\mathit{lhs} : \mathbb{P}\ \mathit{WME} \\
\hline
\operatorname{dom} \mathit{matching} = (\mathit{production}.\mathit{lhs} \cap \operatorname{ran} \mathit{PositiveCondition}) \\
\mathit{lhs} = \mathit{first} (\!| \operatorname{ran} \mathit{matching} |\!) \\
\operatorname{dom} \mathit{binding} = \\
\quad \bigcup(\mathit{ConditionsPositiveComponents} (\!| \mathit{production}.\mathit{lhs} \cap \operatorname{ran} \mathit{PositiveCondition} |\!)) \\
\qquad \cap\, \mathit{Variable} \\
\forall c : \operatorname{dom} \mathit{matching} \bullet \\
\quad \mathit{Matches}_{WME}^{WMETest} (\mathit{binding}, \mathit{PositiveCondition}^{\sim}(c), \mathit{first}(\mathit{matching}(c))) \\
\hline
\end{array}
$$

makeMatch constructs a new match given the production and the matching between the positive conditions and the working memory elements that they matched. The match's binding and left hand side are uniquely determined by the production and the matching, so they are not required arguments to construct a match.

$$
\begin{array}{l}
\mathit{makeMatch} : \\
\quad SP \times (\operatorname{ran} \mathit{PositiveCondition} \rightarrowtail (\mathit{WME} \times \mathbb{N})) \rightarrow \mathit{Match} \\
\hline
\forall\, \mathit{production} : SP; \\
\quad \mathit{matching} : (\operatorname{ran} \mathit{PositiveCondition} \rightarrowtail (\mathit{WME} \times \mathbb{N})) \bullet \\
\quad \exists_1\, \mathit{binding} : \mathit{Binding};\ \mathit{lhs} : \mathbb{P}\ \mathit{WME} \mid \mathit{Match} \bullet \\
\qquad \mathit{makeMatch}(\mathit{production}, \mathit{matching}) = \theta \mathit{Match}
\end{array}
$$

5.2.3 Matching against Working Memory

The definition of match checks the correspondence between the positive conditions and the working memory elements that they matched, but it makes no statement that these elements must be in working memory. Nor does it require that the negative conditions do not have any matches in working memory. We rectify this by constraining how a condition matches against the contents of working working memory

$Matches^{Condition}_{WM}$ takes a binding, a condition and a set of working memory elements, and checks if the condition matches working memory. A positive condition matches working memory if there is a element in working memory that its *WMETest* matches under the binding. A negative condition matches against working memory if there is no way to consistently extend the binding to cover the variables of the negated condition and then find an element in working memory whose test matches. A negative conjunctive condition matches working memory if there is no way to extend the binding and to match all of its sub-conditions against working memory.

$$
\begin{array}{l}
Matches^{Condition}_{WM}\ \_ : \mathbb{P}(Binding \times Condition \times \mathbb{P}\ WME) \\
\hline
\forall\, b : Binding;\ pc : \operatorname{ran} PositiveCondition;\ WM : \mathbb{P}\ WME \bullet \\
\quad Matches^{Condition}_{WM}\ (b, pc, WM) \Leftrightarrow \\
\qquad (\exists\, w : WM \bullet \\
\qquad\quad Matches^{WMETest}_{WME}\ (b, PositiveCondition^{\sim}(pc), w)) \\
\forall\, b : Binding;\ nc : \operatorname{ran} NegativeCondition;\ WM : \mathbb{P}\ WME \bullet \\
\quad Matches^{Condition}_{WM}\ (b, nc, WM) \Leftrightarrow \\
\qquad \neg\, (\exists\, nb : Binding \mid \\
\qquad\quad nb\ ConsistentExtension\ b \land \\
\qquad\quad nb\ Covers\ (ConditionsComponents(nc) \cap Variable) \bullet \\
\qquad\qquad (\exists\, w : WM \bullet \\
\qquad\qquad\quad Matches^{WMETest}_{WME}\ (nb, NegativeCondition^{\sim}(nc), w))) \\
\forall\, b : Binding;\ ncc : \operatorname{ran} NegativeConjunctiveCondition;\ WM : \mathbb{P}\ WME \bullet \\
\quad Matches^{Condition}_{WM}\ (b, ncc, WM) \Leftrightarrow \\
\qquad \neg\, (\exists\, nb : Binding \mid \\
\qquad\quad nb\ ConsistentExtension\ b \land \\
\qquad\quad nb\ Covers\ (ConditionsComponents(ncc) \cap Variable) \bullet \\
\qquad\qquad \forall\, c : NegativeConjunctiveCondition^{\sim}(ncc) \bullet \\
\qquad\qquad\quad Matches^{Condition}_{WM}\ (nb, c, WM))
\end{array}
$$

A match is satisfied by working memory if and only if all of its conditions match against working memory under the match's binding, and the elements they match have the current working memory numbering, i.e., the elements are the copies that are currently in memory.

$$Matches^{Match}_{WM}\ \_ : \mathbb{P}(Match \times \mathbb{P}\ WME \times (WME \rightarrowtail \mathbb{N}))$$

$$\begin{array}{l}
\forall\, m : Match;\ WM : \mathbb{P}\ WME \bullet \\
\quad Matches^{Match}_{WM}(m, WM, WM\#) \Leftrightarrow \\
\qquad (\forall\, c : m.production.lhs \bullet \\
\qquad\quad Matches^{Condition}_{WM}(m.binding, c, WM) \wedge \\
\qquad\quad WM\#(m.matching(c)) = second(m.matching(c)))
\end{array}$$

5.3 Instantiation

When Soar finds a match for a production, it instantiates it and adds it to instantiation memory. The act of instantiation generates an instantiated version of the match's production's conditions and a set of preferences from the match's production's right hand side makes. Soar adds the support conferred by the instantiation to each preference's entry in support memory. Soar checks the support of preferences to decide how to change preference memory. Soar then passes the preference memory changes to decide, which changes the contents of working memory to reflect the new contents of preference memory.

This section defines how a match's production's conditions are instantiated, and then how a match's production's right hand side is instantiated, and then the instantiation data structure schema.

5.3.1 Instantiating a Match's Production's Conditions

This section defines, in bottom-up order, seven recursive functions that instantiate the conditions of a match's production using the match's binding. The end result is a partial function, called a matching, that maps the conditions of the production to their corresponding instantiated condition.

InstantiateVariableOrConstant returns a variable's binding, if it is bound, or returns the variable, if it is unbound, and otherwise returns a constant.

$$\begin{array}{l}
InstantiateVariableOrConstant : \\
\quad (Variable \cup Constant) \times Binding \rightarrow Symbol
\end{array}$$

$$\begin{array}{l}
\forall\, cv : Constant \cup Variable;\ b : Binding \bullet \\
\quad InstantiateVariableOrConstant(cv, b) = ((\mathrm{id}\ \Sigma) \oplus b)(cv)
\end{array}$$

InstantiateSimpleTest returns tests with no arguments unmodified. Equality, same type and not tests, it destructures, instantiates their variable or constant and returns.

$$
\begin{array}{l}
\mathit{InstantiateSimpleTest} : \mathit{SimpleTest} \times \mathit{Binding} \rightarrow \mathit{SimpleTest} \\
\hline
\forall\, t : \mathit{SimpleTest};\ b : \mathit{Binding} \bullet \\
\quad (((t = \mathit{BlankTest} \lor t = \mathit{YesTest} \lor t = \mathit{NoTest} \lor t \in \operatorname{ran} \mathit{DisjunctiveTest}) \\
\qquad \Rightarrow \mathit{InstantiateSimpleTest}(t, b) = t) \land \\
\quad (t \in \operatorname{ran} \mathit{EqualityTest} \Rightarrow \\
\qquad \mathit{InstantiateSimpleTest}(t, b) = \\
\qquad\quad \mathit{EqualityTest}(\mathit{InstantiateVariableOrConstant}(\mathit{EqualityTest}^{\sim}(t), b))) \land \\
\quad (t \in \operatorname{ran} \mathit{SameTypeTest} \Rightarrow \\
\qquad \mathit{InstantiateSimpleTest}(t, b) = \\
\qquad\quad \mathit{SameTypeTest}(\mathit{InstantiateVariableOrConstant}(\mathit{SameTypeTest}^{\sim}(t), b))) \land \\
\quad (t \in \operatorname{ran} \mathit{NotTest} \Rightarrow \\
\qquad \mathit{InstantiateSimpleTest}(t, b) = \\
\qquad\quad \mathit{NotTest}(\mathit{InstantiateVariableOrConstant}(\mathit{NotTest}^{\sim}(t), b))))
\end{array}
$$

InstantiateSetOfSimpleTests iterates the simple test instantiator across a set of tests.

$$
\begin{array}{l}
\mathit{InstantiateSetOfSimpleTests} : \\
\quad \mathbb{P}\ \mathit{SimpleTest} \times \mathit{Binding} \rightarrow \mathbb{P}\ \mathit{SimpleTest} \\
\hline
\forall\, S : \mathbb{P}\ \mathit{SimpleTest};\ b : \mathit{Binding} \bullet \\
\quad ((S = \varnothing \Rightarrow \mathit{InstantiateSetOfSimpleTests}(S, b) = \varnothing) \land \\
\quad (S \neq \varnothing \Rightarrow \\
\qquad (\exists\, s : S \bullet \\
\qquad\quad \mathit{InstantiateSetOfSimpleTests}(S, b) = \\
\qquad\qquad \{\mathit{InstantiateSimpleTest}(s, b)\} \cup \\
\qquad\qquad \mathit{InstantiateSetOfSimpleTests}(S \setminus \{s\}, b))))
\end{array}
$$

InstantiateTest instantiates a test by calling either instantiate simple test, or by unpacking a conjunctive test, instantiating its component tests and repacking.

$$
\begin{array}{|l}
\mathit{InstantiateTest} : \mathit{Test} \times \mathit{Binding} \rightarrow \mathit{Test} \\
\hline
\forall t : \mathit{Test};\ b : \mathit{Binding} \bullet \\
\quad ((t \notin \operatorname{ran} \mathit{ConjunctiveTest} \Rightarrow \\
\qquad \mathit{InstantiateTest}(t, b) = \\
\qquad\quad \mathit{InstantiateSimpleTest}(t, b)) \land \\
\quad (t \in \operatorname{ran} \mathit{ConjunctiveTest} \Rightarrow \\
\qquad \mathit{InstantiateTest}(t, b) = \\
\qquad\quad \mathit{ConjunctiveTest}(\\
\qquad\qquad \mathit{InstantiateSetOfSimpleTests}(\mathit{ConjunctiveTest}^{\sim}(t), b))))
\end{array}
$$

InstantiateWMETest instantiates a working memory element test by instantiating its component parts, and giving the new test the same context acceptable preference test.

$$
\begin{array}{|l}
\mathit{InstantiateWMETest} : \mathit{WMETest} \times \mathit{Binding} \rightarrow \mathit{WMETest} \\
\hline
\forall w : \mathit{WMETest};\ b : \mathit{Binding} \bullet \\
\quad \mathit{InstantiateWMETest}(w, b) = \\
\qquad (\mu\ \mathit{id} : \{\mathit{InstantiateTest}(w.\mathit{id}, b)\} \bullet \\
\qquad\quad (\mu\ \mathit{attribute} : \{\mathit{InstantiateTest}(w.\mathit{attribute}, b)\} \bullet \\
\qquad\qquad (\mu\ \mathit{value} : \{\mathit{InstantiateTest}(w.\mathit{value}, b)\} \bullet \\
\qquad\qquad\quad (\mu\ \mathit{WMETest} \mid \\
\qquad\qquad\qquad \mathit{context\_acceptable\_preference} = \\
\qquad\qquad\qquad\quad w.\mathit{context\_acceptable\_preference} \bullet \\
\qquad\qquad\qquad\qquad \theta\,\mathit{WMETest}))))
\end{array}
$$

InstantiateCondition and *InstantiateSetOfConditions* recurse through conditions instantiating their component working memory element tests or conditions. Z requires that they are defined in a single axiomatic definition to allow them to be mutually recursive.

$$
\begin{array}{l}
\mathit{InstantiateCondition} : \mathit{Condition} \times \mathit{Binding} \rightarrow \mathit{Condition} \\
\mathit{InstantiateSetOfConditions} : \mathbb{P}\ \mathit{Condition} \times \mathit{Binding} \rightarrow \mathbb{P}\ \mathit{Condition} \\
\hline
\forall\, c : \mathit{Condition};\ b : \mathit{Binding} \bullet \\
\quad ((c \in \operatorname{ran} \mathit{PositiveCondition} \Rightarrow \\
\qquad \mathit{InstantiateCondition}(c, b) = \\
\qquad\quad \mathit{PositiveCondition}(\mathit{InstantiateWMETest}(\mathit{PositiveCondition}^{\sim}(c), b))) \wedge \\
\quad (c \in \operatorname{ran} \mathit{NegativeCondition} \Rightarrow \\
\qquad \mathit{InstantiateCondition}(c, b) = \\
\qquad\quad \mathit{NegativeCondition}(\mathit{InstantiateWMETest}(\mathit{NegativeCondition}^{\sim}(c), b))) \wedge \\
\quad (c \in \operatorname{ran} \mathit{NegativeConjunctiveCondition} \Rightarrow \\
\qquad \mathit{InstantiateCondition}(c, b) = \\
\qquad\quad \mathit{NegativeConjunctiveCondition} \\
\qquad\qquad (\mathit{InstantiateSetOfConditions} \\
\qquad\qquad\quad (\mathit{NegativeConjunctiveCondition}^{\sim}(c), b)))) \\
\forall\, S : \mathbb{P}\ \mathit{Condition};\ b : \mathit{Binding} \bullet \\
\quad ((S = \emptyset \Rightarrow \mathit{InstantiateSetOfConditions}(S, b) = \emptyset) \wedge \\
\quad (S \neq \emptyset \Rightarrow \\
\qquad (\exists\, s : S \bullet \\
\qquad\quad \mathit{InstantiateSetOfConditions}(S, b) = \\
\qquad\qquad \{\mathit{InstantiateCondition}(s, b)\} \cup \\
\qquad\qquad\quad \mathit{InstantiateSetOfConditions}(S \setminus \{s\}, b))))
\end{array}
$$

InstantiateConditionsWME instantiates a condition that corresponds to the working memory element that a positive condition matched.

$$
\begin{array}{|l}
\mathit{InstantiateConditionsWME} : \mathit{WME} \rightarrow \mathit{Condition} \\
\hline
\forall\, w : \mathit{WME} \bullet \\
\quad ((w.\mathit{context\_acceptable\_preference} = \mathit{Yes} \Rightarrow \\
\qquad \mathit{InstantiateConditionsWME}(w) = \\
\qquad\quad \mathit{PositiveCondition}(\\
\qquad\qquad \mathit{makeWMETest}(\mathit{EqualityTest}(w.\mathit{id}), \mathit{EqualityTest}(w.\mathit{attribute}), \\
\qquad\qquad\quad \mathit{EqualityTest}(w.\mathit{value}), \mathit{YesTest}))) \\
\wedge \\
\quad (w.\mathit{context\_acceptable\_preference} = \mathit{Yes} \Rightarrow \\
\qquad \mathit{InstantiateConditionsWME}(w) = \\
\qquad\quad \mathit{PositiveCondition}(\\
\qquad\qquad \mathit{makeWMETest}(\mathit{EqualityTest}(w.\mathit{id}), \mathit{EqualityTest}(w.\mathit{attribute}), \\
\qquad\qquad\quad \mathit{EqualityTest}(w.\mathit{value}), \mathit{NoTest}))))
\end{array}
$$

InstantiateConditionMatching instantiates a set of conditions under a binding to produce a partial function that maps each of the conditions to their instantiation. The positive conditions are mapped to a simple instantiated version of the working memory element that they matched. It contains an equality test for each value in the element and the appropriate context acceptable preference test. The negative conditions are recursively instantiated to retain their test structures, but have their variables replaced with identifiers. This retains maximal information about their matching which can be utilized when the chunk is built.

$$
\begin{array}{|l}
\mathit{InstantiateConditionMatching} : \\
\quad (\operatorname{ran} \mathit{PositiveCondition} \rightarrowtail (\mathit{WME} \times \mathbb{N})) \times \\
\quad \mathbb{P}(\mathit{Condition} \setminus \operatorname{ran} \mathit{PositiveCondition}) \times \mathit{Binding} \\
\qquad \rightarrowtail (\mathit{Condition} \rightarrowtail \mathit{Condition}) \\
\hline
\forall\, \mathit{CM} : (\operatorname{ran} \mathit{PositiveCondition} \rightarrowtail (\mathit{WME} \times \mathbb{N}));\ C : \mathbb{P}\ \mathit{Condition};\ b : \mathit{Binding} \bullet \\
\quad ((\mathit{CM} = \emptyset \wedge C = \emptyset \Rightarrow \mathit{InstantiateConditionMatching}(C, b) = \emptyset) \wedge \\
\quad (\mathit{CM} \neq \emptyset \Rightarrow \\
\quad\ (\exists\, \mathit{cwn} : \mathit{CM} \bullet \\
\qquad \mathit{InstantiateConditionMatching}(\mathit{CM}, C, b) = \\
\qquad\quad \{(\mathit{first}(\mathit{cwn}) \longrightarrow \mathit{InstantiateConditionsWME}(\mathit{first}(\mathit{second}(\mathit{cwn}))\}\cup \\
\qquad\qquad \mathit{InstantiateConditionMatching}(\mathit{CM} \setminus \mathit{cwm}, C, b))) \wedge \\
\quad (\mathit{CM} = \emptyset \wedge C \neq \emptyset \Rightarrow \\
\quad\ (\exists\, c : C \bullet \\
\qquad \mathit{InstantiateConditionMatching}(\mathit{CM}, C, b) = \\
\qquad\quad \{(c \mapsto \mathit{InstantiateNegativeCondition}(c, b))\}\cup \\
\qquad\qquad \mathit{InstantiateConditionMatching}(\emptyset, C \setminus \{c\}, b)))
\end{array}
$$

5.3.2 Instantiating the RHS of a Match's Production

MakesPreference takes a binding and produces a function that translates a make into a preference using the binding. The different argument format allows the function to be easily imaged across a set. The binding must cover all of the variables of an argument preference, or *MakesPreference* is undefined.

$$
\begin{array}{l}
\mathit{MakesPreference} : \mathit{Binding} \rightarrow (\mathit{Make} \rightarrow \mathit{Preference}) \\
\hline
\forall\, b : \mathit{Binding} \bullet \forall\, m : \mathit{Make} \bullet \\
\quad \exists\, o : \Sigma \rightarrow \Sigma \mid o = (\mathrm{id}\ \mathit{Constant} \oplus b) \bullet \\
\qquad ((m \in \mathrm{ran}\ \mathit{UnaryM} \Rightarrow \\
\qquad\quad (\exists\, up : \mathit{UnaryPreference} \mid up = \mathit{UnaryM}^{\sim}(m) \bullet \\
\qquad\qquad \mathit{MakesPreference}(b)(m) = \\
\qquad\qquad\quad \mathit{makeUnaryPreference}(o(up.\mathit{id}), o(up.\mathit{attribute}), \\
\qquad\qquad\qquad o(up.\mathit{value}), up.\mathit{preference}))) \wedge \\
\qquad (m \in \mathrm{ran}\ \mathit{BinaryM} \Rightarrow \\
\qquad\quad (\exists\, bp : \mathit{BinaryPreference} \mid bp = \mathit{BinaryM}^{\sim}(m) \bullet \\
\qquad\qquad \mathit{MakesPreference}(b)(m) = \\
\qquad\qquad\quad \mathit{makeBinaryPreference}(o(bp.\mathit{id}), o(bp.\mathit{attribute}), o(bp.\mathit{value}), \\
\qquad\qquad\qquad bp.\mathit{preference}, o(bp.\mathit{referent})))))
\end{array}
$$

5.3.3 Instantiation

An instantiation contains a match, a binding and a matching. The binding is a consistent extension of the match's binding that covers all of the variables of the production's right hand side. The instantiation's right hand side is the set of preferences made from instantiating the production's makes under the instantiation's binding.

The instantiation also contains a matching. The matching is a correspondence between *all* of the conditions of the production and new conditions. The new conditions are instantiated copies of the production's conditions under the match's binding; these record for chunking how the conditions of the productions matched.

Instantiation

$match : Match$
$binding : Binding$
$matching : Condition \nrightarrow Condition$
$lhs : \mathbb{P}\ Condition$
$rhs : \mathbb{P}\ Preference$

$binding\ ConsistentExtension\ match.binding$

$$\begin{aligned}&\mathrm{dom}\ binding = \\ &\quad (\bigcup(MakesComponents(\!|match.production.rhs|\!)) \cap Variable) \\ &\qquad \cup\ \mathrm{dom}\ match.binding\end{aligned}$$

$rhs = (MakesPreference(binding))(\!|match.production.rhs|\!)$

$$\begin{aligned}&matching = \\ &InstantiateConditionMatching(match.matching, \\ &\quad match.production.lhs \setminus \mathrm{ran}\ PositiveCondition, match.binding)\end{aligned}$$

$lhs = \mathrm{ran}\ matching$

makeInstantiation constructs a new instantiation data structure schema given a match and a condition matching. The instantiation's binding, left hand side and right hand side are all uniquely determined from the match and the condition matching.

$makeInstantiation : Match \times (Condition \nrightarrow Condition) \rightarrow Instantiation$

$$\begin{aligned}&\forall\ match : Match;\ matching : (Condition \nrightarrow Condition) \bullet \\ &\qquad \exists_1\ binding : Binding;\ lhs : \mathbb{P}\ Condition;\ rhs : \mathbb{P}\ Preference \mid Instantiation \bullet \\ &\quad makeInstantiation(match, matching) = \theta Instantiation\end{aligned}$$

5.4 Goal Operations and a Match's Goal

Recognition memory reads the contents of goal memory and working memory to determine the support that instantiations confer on preferences. The goal stack is defined in Chapter 7, but is represented in working memory elements. This section defines operations that read from working memory the state, operator and ancestors of a goal, and determine the newest goal that a match matches against.

The state of a goal is the value of the single working memory element of the goal that has attribute state.

$$
\begin{array}{l}
\mathit{GoalState} : \mathit{Identifier} \times \mathbb{P}\ \mathit{TME} \rightarrow \mathit{Identifier} \\
\hline
\forall g : \mathit{Identifier};\ \mathit{TM} : \mathbb{P}\ \mathit{TME} \bullet \\
\quad \exists_1\, t : \mathit{TM} \cap \operatorname{ran} \mathit{WMETME} \bullet \\
\quad\quad \exists w : \mathit{WME} \mid w = \mathit{WMETME}^{\sim}(t) \land \\
\quad\quad\quad w.\mathit{id} = g \land w.\mathit{attribute} = \text{`STATE'} \land \\
\quad\quad\quad w.\mathit{context\_acceptable\_preference} = \mathit{No} \\
\quad\quad \bullet\ \mathit{GoalState}(g, \mathit{TM}) = w.\mathit{value}
\end{array}
$$

The operator of a goal is the value of the single working memory element of the goal that has attribute operator.

$$
\begin{array}{l}
\mathit{GoalOperator} : \mathit{Identifier} \times \mathbb{P}\ \mathit{TME} \rightarrow \mathit{Identifier} \\
\hline
\forall g : \mathit{Identifier};\ \mathit{TM} : \mathbb{P}\ \mathit{TME} \bullet \\
\quad \exists t : \mathit{TM} \cap \operatorname{ran} \mathit{WMETME} \bullet \\
\quad\quad \exists w : \mathit{WME} \mid w = \mathit{WMETME}^{\sim}(t) \land \\
\quad\quad\quad w.\mathit{id} = g \land w.\mathit{attribute} = \text{`OPERATOR'} \land \\
\quad\quad\quad w.\mathit{context\_acceptable\_preference} = \mathit{No} \\
\quad\quad \bullet\ \mathit{GoalOperator}(g, \mathit{TM}) = w.\mathit{value}
\end{array}
$$

GoalAncestors returns the set of identifiers of ancestor goals of a goal.

$$
\begin{array}{l}
\mathit{GoalAncestors} : \mathit{Identifier} \times \mathbb{P}\ \mathit{TME} \rightarrow \mathbb{P}\ \mathit{Identifier} \\
\hline
\forall g : \mathit{Identifier};\ \mathit{TM} : \mathbb{P}\ \mathit{TME} \bullet \\
\quad \mathit{GoalAncestors}(\text{`NIL'}, \mathit{TM}) = \varnothing \land \\
\quad\quad (\exists g_2 : \mathit{Identifier} \mid \mathit{WMETME}(\mathit{makeWME}(g, \text{`OBJECT'}, g_2, \mathit{No})) \in \mathit{TM} \bullet \\
\quad\quad \mathit{GoalAncestors}(g, \mathit{TM}) = \{g_2\} \cup \mathit{GoalAncestors}(g_2, \mathit{TM}))
\end{array}
$$

The support an instantiation confers depends upon the deepest of the goals that it matched, called the match's goal. Soar constrains its productions to match each root variable with an identifier of a goal or attribute impasse. A root variable is one that appears only in the identifier field of the production's conditions.

$$
\begin{array}{|l}
\mathit{MatchGoals} : \mathit{Match} \rightarrow \mathbb{P}\ \mathit{Identifier} \\
\mathit{MatchGoal} : \mathit{Match} \times \mathbb{P}\ \mathit{TME} \rightarrow \mathit{Identifier} \\
\hline
\forall\, m : \mathit{Match} \bullet \\
\quad \mathit{MatchGoals}(m) = m.\mathit{binding}(\!| m.\mathit{production.roots} |\!) \\
\forall\, m : \mathit{Match};\ \mathit{TM} : \mathbb{P}\ \mathit{TME} \bullet \\
\quad \exists\, g : \mathit{MatchGoals}(m) \mid \\
\qquad \neg\, (\exists\, g_2 : \mathit{MatchGoals}(m) \bullet g_2 \neq g \wedge g \in \mathit{GoalAncestors}(g_2, \mathit{TM})) \bullet \\
\qquad \mathit{MatchGoal}(m, \mathit{TME}) = g
\end{array}
$$

5.5 Chunking

Chunking is Soar's explanation based learning mechanism [RL86]. It watches and records some of recognition memory's and decide's operation into two trace memories. When decide fills a slot in working memory, it augments the working memory trace, *TrW*. *TrW* maps each working memory element instance to the require or acceptable preference for the slot and value. After recognition memory instantiates a match, it passes the instantiation to chunking. Chunking creates a data structure called a trace from the instantiation. The trace records the working memory element instances that were matched in the form of instantiated conditions. The instantiated conditions are split into three groups, called grounds, potentials and locals. Each preference instance of the instantiation is marked as dependent upon this trace in the preference memory trace, *TrP*.

After chunking has created the instantiation's trace, it determines if the instantiation augments the match goal's parent with new preferences. If so, the results are used as a starting point for a search through the trace memories. The trace generated conditions in this search are collected and variabilized to produce a new left hand side for a production, and the results are variabilized to produce a new right hand side.

Chunking is presented in eleven sections.

- Section 5.5.1 Extracting Not Tests — instantiations test values from their working memory elements for inequality. The resulting chunks must also check these inequalities, so we extract the not tests of each instantiation, store them in the traces and re-introduce them in Section 5.5.3 Not-ify.
- Section 5.5.2 Variabilize — this section generalizes the sets of instantiated

conditions that chunking finds into conditions by assigning variables to each identifier.

- Section 5.5.3 Not-ify — re-introduces not tests.
- Section 5.5.4 Traces — specifies what is in a trace and provides utilities for constructing them.
- Section 5.5.5 Chunking's State — this section defines chunking's state schema.
- Section 5.5.6 Changing, Initializing and Resetting Chunking — we define constraints on each step of chunking, as well as the standard initialization and resetting for chunking.
- Section 5.5.7 Defining Results — results are the preferences that an instantiation adds to its parent goal, which are also the starting point for chunking's backtracing.
- Section 5.5.8 Chunking an Instantiation — this section provides the operation to build the trace for an instantiation, and to start the search back through the trace memories.
- Section 5.5.9 Tracing Seeds, Locals and Grounded Potentials — this section specifies the basic tracing operations.
- Section 5.5.10 Tracing UnGrounded Potentials — this section defines how to trace a special type of condition that may or may not be included in the chunk's condition.
- Section 5.5.11 ChunkSP — this section specifies how chunking builds the new chunk when backtracing has finished.

5.5.1 Extracting Not Tests

In this section, we specify how to extract the nots from a condition. The nots are doubletons of identifiers that the condition constrained to be different. The presentation is bottom up, starting with a not extractor for tests, then working memory element tests and then condition's nots.

$$Not == \{c_1, c_2 : \mathit{Identifier} \mid c_1 \neq c_2 \bullet \{c_1, c_2\}\}$$

Most types of tests have no nots, but a *NotTest* that matched against two identifiers generates a not doubleton.

$$
\begin{array}{l}
\mathit{TestsNots} : \mathit{Binding} \times \mathit{Test} \times \mathit{Test} \rightarrow \mathbb{P}\ \mathit{Not} \\
\hline
\forall\, b : \mathit{Binding};\ t, vt : \mathit{Test} \bullet \\
\quad ((t \in \{\mathit{BlankTest}, \mathit{YesTest}, \mathit{NoTest}\} \Rightarrow \mathit{TestsNots}(b, t, vt) = \varnothing) \wedge \\
\quad (t \in \operatorname{ran} \mathit{EqualityTest} \Rightarrow \mathit{TestsNots}(b, t, vt) = \varnothing) \wedge \\
\quad (t \in \operatorname{ran} \mathit{NotTest} \Rightarrow \\
\qquad (\exists\, s : \Sigma \mid s = \mathit{NotTest}^{\sim}(t) \bullet \\
\qquad ((((s \in \mathit{Variable}) \wedge (((\mathrm{id}\ \mathit{Constant}) \oplus b)(s) \in \mathit{Identifier}) \\
\qquad\quad \wedge (\mathit{EqualityTest}^{\sim}(vt) \in \mathit{Identifier})) \Rightarrow \\
\qquad\quad \mathit{TestsNots}(b, t, vt) = \{\{((\mathrm{id}\ \mathit{Constant}) \oplus b)(s), \mathit{EqualityTest}^{\sim}(vt)\}\}) \wedge \\
\qquad (((s \in \mathit{Variable}) \wedge (((\mathrm{id}\ \mathit{Constant}) \oplus b)(s) \in \mathit{Identifier}) \\
\qquad\quad \wedge (\mathit{EqualityTest}^{\sim}(vt) \in \mathit{Identifier})) \Rightarrow \\
\qquad\quad \mathit{TestsNots}(b, t, vt) = \varnothing)))) \wedge \\
\quad (t \in \operatorname{ran} \mathit{SameTypeTest} \Rightarrow \mathit{TestsNots}(b, t, vt) = \varnothing) \wedge \\
\quad (t \in \operatorname{ran} \mathit{DisjunctiveTest} \Rightarrow \mathit{TestsNots}(b, t, vt) = \varnothing) \wedge \\
\quad (t \in \operatorname{ran} \mathit{ConjunctiveTest} \Rightarrow \\
\qquad \mathit{TestsNots}(b, t, vt) = \\
\qquad\quad \bigcup((\lambda\, t : \mathit{Test} \bullet \mathit{TestsNots}(b, t, vt))(\!|\mathit{ConjunctiveTest}^{\sim}(t)|\!)))
\end{array}
$$

The nots of a working memory element test is the set of nots in its id, attribute and value tests.

$$
\begin{array}{l}
\mathit{WMETestsNots} : \mathit{Binding} \times \mathit{WMETest} \times \mathit{WMETest} \rightarrow \mathbb{P}\ \mathit{Not} \\
\hline
\forall\, b : \mathit{Binding};\ wt, w : \mathit{WMETest} \bullet \\
\quad \mathit{WMETestsNots}(b, wt, w) = \\
\qquad \mathit{TestsNots}(b, wt.id, w.id) \cup \mathit{TestsNots}(b, wt.attribute, w.attribute) \\
\qquad \cup\ \mathit{TestsNots}(b, wt.value, w.value)
\end{array}
$$

We only extract nots from positive conditions, as the nots inside of a negative condition are known not to have matched against any values. The nots of a positive condition are the nots of its working memory element tests.

$$
\begin{array}{l}
\mathit{ConditionsNots} : \mathit{Binding} \times \mathit{Condition} \times \mathit{Condition} \rightarrow \mathbb{P}\ \mathit{Not} \\
\hline
\forall\, b : \mathit{Binding};\ c, cm : \mathit{Condition} \bullet \\
\quad ((c \in \operatorname{ran} \mathit{PositiveCondition} \Rightarrow \\
\qquad \mathit{ConditionsNots}(b, c, cm) = \\
\qquad\quad \mathit{WMETestsNots}(b, \mathit{PositiveCondition}^{\sim}(c), \\
\qquad\quad \mathit{PositiveCondition}^{\sim}(cm))) \wedge \\
\quad (c \notin \operatorname{ran} \mathit{PositiveCondition} \Rightarrow \\
\qquad \mathit{ConditionsNots}(b, c, cm) = \varnothing))
\end{array}
$$

5.5.2 Variabilize

Variabilize generalizes Chunking's instantiated conditions by replacing their identifiers with variables. After chunking has finished its search through the trace memory, it picks an assignment, a partial injection from identifiers to variables, that covers all of the variables of its condition. It recurses down into the tests of its conditions, replacing all occurrences of identifiers with their variables.

$$Assignment == Identifier \rightarrowtail Variable$$

A symbol is variabilized by returning the symbol if it is a constant and otherwise by returning its value under the assignment.

$$\begin{array}{l}
VariabilizeSymbol : \Sigma \times Assignment \rightarrow \Sigma \\
\hline
\forall c : \Sigma;\ a : Assignment \bullet \\
\quad ((c \in Identifier \Rightarrow \\
\qquad (\exists v : Variable \mid v \notin \operatorname{ran} a \bullet VariabilizeSymbol(c, a) = v)) \wedge \\
\quad (c \notin Identifier \Rightarrow VariabilizeSymbol(c, a) = c))
\end{array}$$

A test is variabilized by variabilizing its component symbols, or returning the test if it is indivisible.

$$\begin{array}{l}
VariabilizeTest : Test \times Assignment \rightarrow Test \\
\hline
\forall t : Test;\ a : Assignment \bullet \\
\quad ((t \in \{BlankTest, YesTest, NoTest\} \cup \operatorname{ran} DisjunctiveTest \\
\qquad \Rightarrow VariabilizeTest(t, a) = t) \wedge \\
\quad (t \in \operatorname{ran} EqualityTest \Rightarrow \\
\qquad VariabilizeTest(t, a) = \\
\qquad\quad EqualityTest(VariabilizeSymbol(EqualityTest^{\sim}(t), a))) \wedge \\
\quad (t \in \operatorname{ran} NotTest \Rightarrow \\
\qquad VariabilizeTest(t, a) = \\
\qquad\quad NotTest(VariabilizeSymbol(NotTest^{\sim}(t), a))) \wedge \\
\quad (t \in \operatorname{ran} SameTypeTest \Rightarrow \\
\qquad VariabilizeTest(t, a) = \\
\qquad\quad SameTypeTest(VariabilizeSymbol(SameTypeTest^{\sim}(t), a))) \wedge \\
\quad (t \in \operatorname{ran} ConjunctiveTest \Rightarrow \\
\qquad VariabilizeTest(t, a) = \\
\qquad\quad ConjunctiveTest(((\lambda\, st : Test \bullet VariabilizeTest(st, a)) \\
\qquad\qquad (\!| ConjunctiveTest^{\sim}(t) |\!)))))
\end{array}$$

A working memory element test is variabilized by variabilizing its id, attribute and value tests, and keeping its acceptable preference test.

$$\begin{array}{l}
\mathit{VariabilizeWMETest} : \mathit{WMETest} \times \mathit{Assignment} \rightarrow \mathit{WMETest} \\
\hline
\forall\, w : \mathit{WMETest};\ a : \mathit{Assignment} \bullet \\
\quad \mathit{VariabilizeWMETest}(w, a) = \\
\quad\quad (\mu\, \mathit{id} : \{\mathit{VariabilizeTest}(w.\mathit{id}, a)\} \bullet \\
\quad\quad\quad (\mu\, \mathit{attribute} : \{\mathit{VariabilizeTest}(w.\mathit{attribute}, a)\} \bullet \\
\quad\quad\quad\quad (\mu\, \mathit{value} : \{\mathit{VariabilizeTest}(w.\mathit{value}, a)\} \bullet \\
\quad\quad\quad\quad\quad (\mu\, \mathit{context\_acceptable\_preference} : \{w.\mathit{context\_acceptable\_preference}\} \bullet \\
\quad\quad\quad\quad\quad \theta\, \mathit{WMETest}))))
\end{array}$$

A condition is variabilized by variabilizing its recursive set of conditions or its component working memory element test.

$$\begin{array}{l}
\mathit{VariabilizeCondition} : \mathit{Condition} \times \mathit{Assignment} \rightarrow \mathit{Condition} \\
\mathit{VariabilizeSetOfConditions} : \\
\quad \mathbb{P}\, \mathit{Condition} \times \mathit{Assignment} \rightarrow \mathbb{P}\, \mathit{Condition} \\
\hline
\forall\, c : \mathit{Condition};\ a : \mathit{Assignment} \bullet \\
\quad ((c \in \operatorname{ran} \mathit{PositiveCondition} \Rightarrow \\
\quad\quad \mathit{VariabilizeCondition}(c, a) = \\
\quad\quad\quad \mathit{PositiveCondition}(\mathit{VariabilizeWMETest}(\mathit{PositiveCondition}^{\sim}(c), a))) \wedge \\
\quad (c \in \operatorname{ran} \mathit{NegativeCondition} \Rightarrow \\
\quad\quad \mathit{VariabilizeCondition}(c, a) = \\
\quad\quad\quad \mathit{NegativeCondition}(\mathit{VariabilizeWMETest}(\mathit{NegativeCondition}^{\sim}(c), a))) \wedge \\
\quad (c \in \operatorname{ran} \mathit{NegativeConjunctiveCondition} \Rightarrow \\
\quad\quad \mathit{VariabilizeCondition}(c, a) = \\
\quad\quad\quad \mathit{NegativeConjunctiveCondition}(\\
\quad\quad\quad\quad \mathit{VariabilizeSetOfConditions}(\mathit{NegativeConjunctiveCondition}^{\sim}(c), a)))) \\
\forall\, C : \mathbb{P}\, \mathit{Condition};\ a : \mathit{Assignment} \bullet \\
\quad ((C = \varnothing \Rightarrow \mathit{VariabilizeSetOfConditions}(C, a) = \varnothing) \wedge \\
\quad (C \neq \varnothing \Rightarrow \\
\quad\quad (\exists\, c : C \bullet \\
\quad\quad\quad \mathit{VariabilizeSetOfConditions}(C, a) = \\
\quad\quad\quad\quad \{\mathit{VariabilizeCondition}(c, a)\} \cup \mathit{VariabilizeSetOfConditions}(C \setminus \{c\}, a))))
\end{array}$$

Chunking cannot produce a set of instantiated conditions to variabilize because it must maintain the correspondence between the conditions and the working memory element instances that they matched. Chunking actually produces a partial function from conditions to working memory element instances. This is variabilized to produce both a matching for the chunk's match and another form for the chunk's instantiation.

$$
\begin{array}{l}
\mathit{VariabilizeConditionMatching} : (\mathit{Condition} \rightarrowtail\!\!\!\!\!\!+\;\; (\mathit{WME} \times \mathbb{N})) \times \mathit{Assignment} \rightarrow \\
\quad ((\mathit{Condition} \rightarrowtail\!\!\!\!\!\!+\;\; (\mathit{WME} \times \mathbb{N})) \times (\mathit{Condition} \rightarrowtail\!\!\!\!\!\!+\;\; \mathit{Condition})) \\
\hline
\forall\, \mathit{mm} : (\mathit{Condition} \rightarrowtail\!\!\!\!\!\!+\;\; (\mathit{WME} \times \mathbb{N}));\ a : \mathit{Assignment} \bullet \\
\quad ((\mathit{mm} = \emptyset \Rightarrow \mathit{VariabilizeConditionMatching}(\mathit{mm}, a) = (\emptyset, \emptyset)) \wedge \\
\quad\quad (\mathit{mm} \neq \emptyset \Rightarrow \\
\quad\quad\quad (\exists\, c : \operatorname{dom} \mathit{mm} \bullet \\
\quad\quad\quad\quad (\exists\, \mathit{cwncc} : \{\mathit{VariabilizeConditionMatching}(\{c\} \lhd \mathit{mm}, a)\} \bullet \\
\quad\quad\quad\quad \mathit{VariabilizeConditionMatching}(\mathit{mm}, a) = \\
\quad\quad\quad\quad\quad (\{\mathit{VariabilizeCondition}(c, a) \mapsto \mathit{mm}(c)\} \cup \mathit{first}(\mathit{cwncc}), \\
\quad\quad\quad\quad\quad \{\mathit{VariabilizeCondition}(c, a) \mapsto c\} \cup \mathit{second}(\mathit{cwncc})))))
\end{array}
$$

The results of an instantiation are variabilized to produce the new right hand side for the chunk. *VariabilizePreferenceToMake* maps a preference to a make. and *VariabilizeRHS* maps a set of preferences to its set of makes.

```
VariabilizePreferenceToMake : Preference × Assignment → Make
────────────────
∀ p : Preference; a : Assignment •
  ((p ∈ ran UnaryP ⇒
    (∃ up : UnaryPreference | up = UnaryP~(p) •
      VariabilizePreferenceToMake(p, a) =
        makeUnaryMake(VariabilizeSymbol(up.id, a),
          VariabilizeSymbol(up.attribute, a),
          VariabilizeSymbol(up.value, a), up.preference))) ∧
  (p ∈ ran BinaryP ⇒
    (∃ bp : BinaryPreference | bp = BinaryP~(p) •
      VariabilizePreferenceToMake(p, a) =
        makeBinaryMake(VariabilizeSymbol(bp.id, a),
          VariabilizeSymbol(bp.attribute, a),
          VariabilizeSymbol(bp.value, a), bp.preference,
            VariabilizeSymbol(bp.referent, a)))))
```

```
VariabilizeRHS : ℙ Preference × Assignment → ℙ Make
────────────────
∀ ps : ℙ Preference; a : Assignment •
  ((ps = ∅ ⇒ VariabilizeRHS(ps, a) = ∅) ∧
    (ps ≠ ∅ ⇒
      (∃ p : ps •
        VariabilizeRHS(ps, a) =
          {VariabilizePreferenceToMake(p, a)} ∪
          VariabilizeRHS(ps \ {p}, a))))
```

5.5.3 Not-ify

When a trace for an instantiation is created, the not extractor saves the set of doubletons of identifiers that the match's condition constrained to be different. As chunking searches through the memory of traces, it unions in the set of not doubletons for each trace that it adds. When chunking is completed, not-ify re-installs these nots into the new conditions.

Not-ify is called after variabilize, so we map the set of doubletons of identifiers to a set of doubletons of variables, called *VNots*. The function *NotsToVNots* maps set of identifer nots to variable nots using the chunk's assignment.

$$VNot == \{v_1, v_2 : Variable \mid v_1 \neq v_2 \bullet \{v_1, v_2\}\}$$

$$\begin{array}{|l}
NotsToVNots : \mathbb{P}\ Not \times Assignment \rightarrow \mathbb{P}\ VNot \\
\hline
\forall N : \mathbb{P}_1\ Not;\ a : Assignment \bullet \\
\quad NotsToVNots(N, a) = \\
\qquad \{n : N;\ i, j : Identifier \mid i \neq j \wedge n = \{i, j\} \bullet \{a(i), a(j)\}\}
\end{array}$$

Not-ify's functions all take three arguments: an object to not-ify, the set of variable nots to install and the set of variables bound in the walk through the conditions. As not-ify walks down the condition matchings, into the conditions, working memory element tests and tests, it adds the variables that the conditions have bound to the set of variables. As it descends, it also installs not tests for those variable nots that have had both of their variables bound. Each not-ify function returns the possibly augmented set of bound variables, and the possibly reduced set of variable nots left to be installed.

As the domains and ranges of the not-ify functions must conveniently handle triples of items, we extend the standard Z *first* and *second* to sequences of three elements ([Spi89] 93).

$$\begin{array}{|l}
[X, Y, Z] \\
\hline
first_3 : X \times Y \times Z \rightarrow X \\
second_3 : X \times Y \times Z \rightarrow Y \\
third_3 : X \times Y \times Z \rightarrow Z \\
\hline
\forall x : X;\ y : Y;\ z : Z \bullet first_3((x, y, z)) = x \\
\forall x : X;\ y : Y;\ z : Z \bullet second_3((x, y, z)) = y \\
\forall x : X;\ y : Y;\ z : Z \bullet third_3((x, y, z)) = z
\end{array}$$

NotifyTest returns most tests unmodified, but if a test is an equality for an unbound variable it binds the variable by adding it to the variable set, and recurses to possibly add in an as of yet unassigned variable not. *NotifySetOfTests* sequences through a set of tests in non-deterministic order to install the variable nots.

$$
\begin{array}{l}
\mathit{NotifyTest} : \\
\quad \mathit{Test} \times \mathbb{P}\ \mathit{VNot} \times \mathbb{P}\ \mathit{Variable} \rightarrow \\
\quad \mathit{Test} \times \mathbb{P}\ \mathit{VNot} \times \mathbb{P}\ \mathit{Variable} \\
\mathit{NotifySetOfTests} : \\
\quad \mathbb{P}\ \mathit{Test} \times \mathbb{P}\ \mathit{VNot} \times \mathbb{P}\ \mathit{Variable} \rightarrow \\
\quad \mathbb{P}\ \mathit{Test} \times \mathbb{P}\ \mathit{VNot} \times \mathbb{P}\ \mathit{Variable} \\
\hline
\forall t : \mathit{Test};\ N : \mathbb{P}\ \mathit{VNot};\ V : \mathbb{P}\ \mathit{Variable} \bullet \\
\quad ((t \in \mathrm{ran}\ \mathit{EqualityTest} \Rightarrow \\
\quad\quad (\mathit{EqualityTest}^{\sim}(t) \notin \mathit{Variable} \Rightarrow \\
\quad\quad\quad \mathit{NotifyTest}(t, N, V) = (t, N, V)) \wedge \\
\quad\quad (\mathit{EqualityTest}^{\sim}(t) \in \mathit{Variable} \Rightarrow \\
\quad\quad\quad (\exists v : \mathit{Variable} \mid v = \mathit{EqualityTest}^{\sim}(t) \bullet \\
\quad\quad\quad\quad ((\exists n : N \bullet \\
\quad\quad\quad\quad\quad v \in n \wedge (n \setminus \{v\}) \cap V \neq \varnothing) \Rightarrow \\
\quad\quad\quad\quad\quad (\exists n : N;\ v_2 : \mathit{Variable} \mid n = \{v, v_2\} \wedge v_2 \in V \bullet \\
\quad\quad\quad\quad\quad\quad (\exists r : \{\mathit{NotifyTest}(\mathit{ConjunctiveTest}(\{t, \mathit{NotTest}(v_2)\}), N \setminus \{n\}, V)\} \bullet \\
\quad\quad\quad\quad\quad\quad \mathit{NotifyTest}(t, N, V) = \mathit{NotifyTest}(\mathit{first}_3(r), \mathit{second}_3(r), \mathit{third}_3(r))))) \wedge \\
\quad\quad\quad\quad \neg\ (\exists n : N \bullet v \in n \wedge (n \setminus \{v\}) \cap V \neq \varnothing) \Rightarrow \\
\quad\quad\quad\quad\quad \mathit{NotifyTest}(t, N, V) = (t, N, V \cup \{v\})))) \wedge \\
\quad (t \in \mathrm{ran}\ \mathit{ConjunctiveTest} \Rightarrow \\
\quad\quad (\exists tNV : \{\mathit{NotifySetOfTests}(\mathit{ConjunctiveTest}^{\sim}(t), N, V)\} \bullet \\
\quad\quad\quad \mathit{NotifyTest}(t, N, V) = (\mathit{ConjunctiveTest}(\mathit{first}_3(tNV)), \mathit{second}_3(tNV), \mathit{third}_3(tNV)))) \wedge \\
\quad (t \notin (\mathrm{ran}\ \mathit{EqualityTest} \cup \mathrm{ran}\ \mathit{ConjunctiveTest}) \Rightarrow \\
\quad\quad \mathit{NotifyTest}(t, N, V) = (t, N, V))) \\
\\
\forall T : \mathbb{P}\ \mathit{Test};\ N : \mathbb{P}\ \mathit{VNot};\ V : \mathbb{P}\ \mathit{Variable} \bullet \\
\quad ((T = \varnothing \Rightarrow \mathit{NotifySetOfTests}(T, N, V) = (T, N, V)) \wedge \\
\quad (T \neq \varnothing \Rightarrow \\
\quad\quad (\exists t : T \bullet \\
\quad\quad\quad (\exists tNV : \{\mathit{NotifyTest}(t, N, V)\} \bullet \\
\quad\quad\quad\quad (\exists TNV : \{\mathit{NotifySetOfTests}(T \setminus \{t\}, \mathit{second}_3(tNV), \mathit{third}_3(tNV))\} \bullet \\
\quad\quad\quad\quad\quad \mathit{NotifySetOfTests}(T, N, V) = \\
\quad\quad\quad\quad\quad\quad (\{t\} \cup \mathit{first}_3(TNV), \mathit{second}_3(TNV), \mathit{third}_3(TNV)))))))
\end{array}
$$

NotifyWMETest sequences through the id, attribute and value to install the variable nots.

$$
\begin{array}{l}
\mathit{NotifyWMETest} : \mathit{WMETest} \times \mathbb{P}\ \mathit{VNot} \times \mathbb{P}\ \mathit{Variable} \to \\
\qquad \mathit{WMETest} \times \mathbb{P}\ \mathit{VNot} \times \mathbb{P}\ \mathit{Variable} \\
\hline
\forall\, w : \mathit{WMETest};\ n : \mathbb{P}\ \mathit{VNot};\ b : \mathbb{P}\ \mathit{Variable} \bullet \\
\quad \mathit{NotifyWMETest}(w, n, b) = \\
\quad\quad (\mu\, \mathit{idnb} : \{\mathit{NotifyTest}(w.\mathit{id}, n, b)\} \bullet \\
\quad\quad\quad (\mu\, \mathit{attributenb} : \{\mathit{NotifyTest}(w.\mathit{attribute}, \mathit{second}_3(\mathit{idnb}), \mathit{third}_3(\mathit{idnb}))\} \bullet \\
\quad\quad\quad\quad (\mu\, \mathit{attribute} : \{\mathit{first}_3(\mathit{attributenb})\} \bullet \\
\quad\quad\quad\quad\quad (\mu\, \mathit{valuenb} : \{\mathit{NotifyTest}(w.\mathit{value}, \mathit{second}_3(\mathit{attributenb}), \mathit{third}_3(\mathit{attributenb}))\} \bullet \\
\quad\quad\quad\quad\quad\quad (\mathit{makeWMETest}(\mathit{first}_3(\mathit{idnb}), \mathit{first}_3(\mathit{attributenb}), \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{first}_3(\mathit{valuenb}), w.\mathit{context\_acceptable\_preference}), \\
\quad\quad\quad\quad\quad\quad\quad \mathit{second}_3(\mathit{valuenb}), \mathit{third}_3(\mathit{valuenb})))))
\end{array}
$$

NotifyCondition unpacks conditions positive conditions to install variable nots, but does not install nots in negated conditions.

$$
\begin{array}{l}
\mathit{NotifyCondition} : \\
\quad \mathit{Condition} \times \mathbb{P}\ \mathit{VNot} \times \mathbb{P}\ \mathit{Variable} \to \mathit{Condition} \times \mathbb{P}\ \mathit{VNot} \times \mathbb{P}\ \mathit{Variable} \\
\hline
\forall\, c : \mathit{Condition};\ n : \mathbb{P}\ \mathit{VNot};\ b : \mathbb{P}\ \mathit{Variable} \bullet \\
\quad ((c \in \operatorname{ran} \mathit{PositiveCondition} \Rightarrow \\
\quad\quad (\exists\, \mathit{pcnb} : \{\mathit{NotifyWMETest}(\mathit{PositiveCondition}^{\sim}(c), n, b)\} \bullet \\
\quad\quad\quad \mathit{NotifyCondition}(c, n, b) = \\
\quad\quad\quad\quad (\mathit{PositiveCondition}(\mathit{first}_3(\mathit{pcnb})), \\
\quad\quad\quad\quad\quad \mathit{second}_3(\mathit{pcnb}), \mathit{third}_3(\mathit{pcnb})))) \land \\
\quad (c \notin \operatorname{ran} \mathit{PositiveCondition} \Rightarrow \\
\quad\quad \mathit{NotifyCondition}(c, n, b) = (c, n, b)))
\end{array}
$$

Finally, *NotifyConditionMatchings* recurses through the two style of condition matchings to install all of the variable nots.

$$\begin{array}{l}
\mathit{NotifyConditionMatchings} : \\
\quad ((\mathit{Condition} \twoheadrightarrow (\mathit{WME} \times \mathbb{N})) \times (\mathit{Condition} \twoheadrightarrow \mathit{Condition})) \times \mathbb{P}\ \mathit{VNot} \times \mathbb{P}\ \mathit{Variable} \rightarrow \\
\quad ((\mathit{Condition} \twoheadrightarrow (\mathit{WME} \times \mathbb{N})) \times (\mathit{Condition} \twoheadrightarrow \mathit{Condition})) \times \mathbb{P}\ \mathit{VNot} \times \mathbb{P}\ \mathit{Variable} \\
\hline
\forall\, cwn : (\mathit{Condition} \twoheadrightarrow (\mathit{WME} \times \mathbb{N}));\ cc : (\mathit{Condition} \twoheadrightarrow \mathit{Condition}); \\
\quad N : \mathbb{P}\ \mathit{VNot};\ V : \mathbb{P}\ \mathit{Variable} \bullet \\
\quad ((cwn = \emptyset \wedge cc = \emptyset \Rightarrow \\
\qquad \mathit{NotifyConditionMatchings}((cwn, cc), N, V) = ((\emptyset, \emptyset), N, V)) \wedge \\
\quad (cwn \neq \emptyset \wedge cc \neq \emptyset \Rightarrow \\
\qquad (\exists\, c : \operatorname{dom} cwn \cap \operatorname{dom} cc \bullet \\
\qquad\quad (\exists\, vcNV : \{\mathit{NotifyCondition}(c, N, V)\} \bullet \\
\qquad\quad (\exists\, cnnccNV : \{\mathit{NotifyConditionMatchings}(\\
\qquad\qquad\quad (\{c\} \lhd cwn, \{c\} \lhd cc), \mathit{second}_3(vcNV), \mathit{third}_3(vcNV))\} \bullet \\
\qquad\qquad \mathit{NotifyConditionMatchings}((cwn, cc), N, V) = \\
\qquad\qquad\quad ((\mathit{first}(\mathit{first}_3(cnnccNV)) \oplus \{\mathit{first}_3(vcNV) \mapsto cwn(c)\}, \\
\qquad\qquad\qquad \mathit{second}(\mathit{first}_3(cnnccNV)) \oplus \{\mathit{first}_3(vcNV) \mapsto c\}), \\
\qquad\qquad\qquad \mathit{second}_3(cnnccNV), \mathit{third}_3(cnnccNV)))))))
\end{array}$$

5.5.4 Traces

Chunking builds a trace for each instantiation starting with the set of instantiated conditions of the instantiation's matching. The trace uses the contents of temporary memory to segregate the conditions into three classes: grounds, potentials and locals. It also records the not doubletons that the match constrained to be different and the instantiation's matches matching.

The grounds of a trace are the instantiated conditions that were directlv matched from a parent goal through a chain of conditions. The potentials are those conditions that are in the transitive closure of a parent of the match goal, but were not directly matched from the parent. All of the non-ground negative conditions are also added into the potentials, as they can not be traced through because they never had a corresponding element in working memory. The locals are the remaining conditions, and should be exactly those positive conditions that are available only from the transitive closure of the match goal.

$$
\begin{array}{l}
\mathit{Grounds} : \mathbb{P}\ \mathit{TME} \times \mathit{Instantiation} \rightarrow \mathbb{P}\ \mathit{Condition} \\
\mathit{Potentials} : \mathbb{P}\ \mathit{TME} \times \mathit{Instantiation} \rightarrow \mathbb{P}\ \mathit{Condition} \\
\mathit{Locals} : \mathbb{P}\ \mathit{TME} \times \mathit{Instantiation} \rightarrow \mathbb{P}(\mathrm{ran}\ \mathit{PositiveCondition}) \\
\mathit{Nots} : \mathit{Instantiation} \rightarrow \mathbb{P}\ \mathit{Not} \\
\hline
\forall\ \mathit{TM} : \mathbb{P}\ \mathit{TME};\ i : \mathit{Instantiation} \bullet \\
\quad \mathit{Grounds}(\mathit{TM}, i) = \\
\quad\quad \mathit{TCI}_{\mathrm{Condition}}(\\
\quad\quad\quad \mathit{OfId}_{\mathrm{Condition}}(\\
\quad\quad\quad\quad \mathit{GoalAncestors}(\mathit{MatchGoal}(i.\mathit{match}, \mathit{TM}), \mathit{TM}), \\
\quad\quad\quad\quad \mathrm{ran}\ i.\mathit{matching}), \mathrm{ran}\ i.\mathit{matching}) \\
\\
\forall\ \mathit{TM} : \mathbb{P}\ \mathit{TME};\ i : \mathit{Instantiation} \bullet \\
\quad \mathit{Potentials}(\mathit{TM}, i) = \\
\quad\quad ((\mathrm{ran}\ i.\mathit{matching} \setminus \mathrm{ran}\ \mathit{PositiveCondition}) \cup \\
\quad\quad \mathit{TCI}_{\mathrm{Condition}}(\\
\quad\quad\quad \mathit{OfId}_{\mathrm{Condition}}(\\
\quad\quad\quad\quad (\mathit{TCI}_{\mathit{TME}}(\mathit{TM}))(\mathit{GoalAncestors}(\mathit{MatchGoal}(i.\mathit{match}, \mathit{TM}), \mathit{TM})), \\
\quad\quad\quad\quad \mathrm{ran}\ i.\mathit{matching}), \qquad \mathrm{ran}\ i.\mathit{matching}) \setminus \mathit{Grounds}(\mathit{TM}, i)) \\
\\
\forall\ \mathit{TM} : \mathbb{P}\ \mathit{TME};\ i : \mathit{Instantiation} \bullet \\
\quad \mathit{Locals}(\mathit{TM}, i) = \mathrm{ran}\ i.\mathit{matching} \setminus (\mathit{Grounds}(\mathit{TM}, i) \cup \mathit{Potentials}(\mathit{TM}, i)) \\
\\
\forall\ i : \mathit{Instantiation} \bullet \\
\quad \mathit{Nots}(i) = \bigcup\{c : \mathrm{dom}\ i.\mathit{matching} \bullet \mathit{ConditionsNots}(i.\mathit{binding}, c, i.\mathit{matching}(c))\}
\end{array}
$$

The *Trace* data schema holds the grounds, potentials, locals, nots and a matching. The matching is a correspondence between the instantiated versions of the positive conditions and the working memory element instances that they matched.

$$
\begin{array}{|l}
\textit{Trace} \\
\hline
\textit{grounds}, \textit{potentials}, \textit{locals} : \mathbb{P}\ \textit{Condition} \\
\textit{nots} : \mathbb{P}\ \textit{Not} \\
\textit{matching} : \textit{Condition} \rightarrowtail (\textit{WME} \times \mathbb{N}) \\
\hline
\operatorname{dom} \textit{matching} = (\textit{grounds} \cup \textit{potentials} \cup \textit{locals}) \cap \operatorname{ran} \textit{PositiveCondition} \\
\hline
\end{array}
$$

The function *makeTrace* is provided to simplify the creation of traces by chunking.

$$
\begin{array}{|l}
\textit{makeTrace} : \mathbb{P}\ \textit{TME} \times \textit{Instantiation} \rightarrow \textit{Trace} \\
\hline
\forall\, \textit{TM} : \mathbb{P}\ \textit{TME};\ i : \textit{Instantiation} \bullet \\
\quad \textit{makeTrace}(\textit{TM}, i) = \\
\quad\quad (\mu\, \textit{grounds} : \{\textit{Grounds}(\textit{TM}, i)\}; \\
\quad\quad\quad \textit{potentials} : \{\textit{Potentials}(\textit{TM}, i)\}; \\
\quad\quad\quad \textit{locals} : \{\textit{Locals}(\textit{TM}, i)\}; \\
\quad\quad\quad \textit{nots} : \{\textit{Nots}(i)\}; \\
\quad\quad\quad \textit{matching} : \{i.\textit{match}.\textit{matching}\} \bullet \\
\quad\quad \theta\, \textit{Trace})
\end{array}
$$

The components of a trace is the components of its conditions, nots and the components of the working memory element instances that it matched.

$$
\begin{array}{|l}
\textit{TracesComponents} : \textit{Trace} \rightarrow \mathbb{P}\ \Sigma \\
\hline
\forall\, t : \textit{Trace} \bullet \\
\quad \textit{TracesComponents}(t) = \\
\quad\quad \bigcup(\textit{ConditionsComponents} ⦇ t.\textit{grounds} \cup t.\textit{potentials} \cup t.\textit{locals} ⦈) \cup \\
\quad\quad \bigcup t.\textit{nots} \cup \\
\quad\quad \bigcup \textit{first} ⦇ \operatorname{ran} \textit{matching} ⦈
\end{array}
$$

5.5.5 Chunking's State

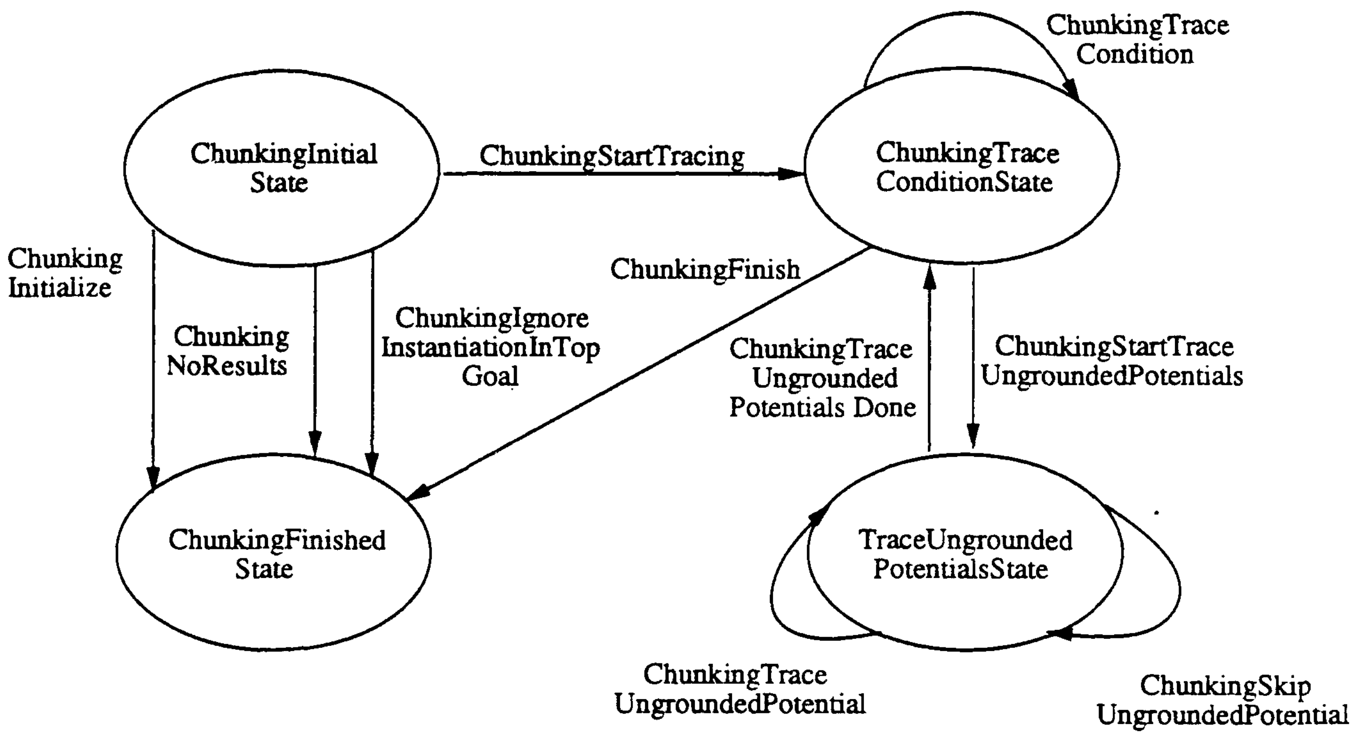
ChunkingTrace Condition
ChunkingInitial State
ChunkingStartTracing
ChunkingTrace ConditionState
Chunking Initialize
Chunking NoResults
ChunkingIgnore InstantiationInTop Goal
ChunkingFinish
ChunkingTrace Ungrounded Potentials Done
ChunkingStartTrace UngroundedPotentials
ChunkingFinished State
TraceUngrounded PotentialsState
ChunkingTrace UngroundedPotential
ChunkingSkip UngroundedPotential

Figure 5.1: Chunking's State Machine

Chunking's state machine has only four states, but has the most complicated state schema of any of Soar's state machines.

- *ChunkingInitialState* — chunking starts in this state.
- *ChunkingTraceConditionState* — traces through the local conditions.
- *ChunkingTraceUnGroundedPotentialsState* — traces through all of the ungrounded potentials, one at a time.
- *ChunkingFinishedState* — chunking finishes in this state.

ChunkingState ::= *ChunkingInitialState*
| *ChunkingTraceConditionState*
| *ChunkingTraceUnGroundedPotentialsState*
| *ChunkingFinishedState*

When chunking processes an instantiation, it may or may not produce a chunk, depending upon the results of the instantiation's right hand side. If it does produce a new chunk, then it returns the new instantiation for that chunk so that preference phase can process it. The *ChunkingResult* type is used to inform the preference phase that chunking is returning a new instantiation.

ChunkingResult ::= *NewInstantiation* | *NoInstantiation*

Soar is actually viewed as continuously learning from its operation. However, in practice Soar's learning mechanism is the most difficult part of Soar to use. Chunking is difficult to use for a variety of reasons: it is rather complicated so users find it difficult to understand, it can create expensive rules which prohibitively degrade performance ([Tam91]), it has some known overgeneralization problems ([LCAS90], 70, 73, 74), and it has yet to be well integrated with our theoretical framework - the problem space computational model ([NYL+91]).

For convenience we allow Soar users to turn learning partially off in two ways. The chunking state schema has a switch *learn* of type *Learn*. When learn is off or if chunking backtraces through a local goal's 'QUIESCENCE' element, a chunk is learned but it is not variabilized and not-ified. These internal chunks act like restricted productions; they can match only one sequence of elements, they confer support on their resulting preferences, and they are guaranteed to be very cheap to match. When an internal chunk's single instantiation retracts, Soar deletes the production from production memory as it is never likely to match again.

Learn ::= *On* | *Off*

Chunking shares six memories with other modules:

- *SPM* — the set of productions
- *TM* — temporary memory
- *WM* — working memory
- *PM* — preference memory
- *GM* — goal memory and
- *IS* — instantiation memory, the set of matches that have been instantiated by the preference phase.

Chunking

$$
\begin{array}{l}
SPM : \mathbb{P}\, SP \\
TM : \mathbb{P}\, TME \\
WM : \mathbb{P}\, WME \\
PM : \mathbb{P}\, Preference \\
GM : \mathbb{P}\, Identifier \\
IS : \mathbb{P}\, Instantiation \\
WM\# : WME \rightarrowtail \mathbb{N} \\
PM\# : Preference \rightarrowtail \mathbb{N} \\
max\_preference, max\_wme : \mathbb{N} \\
TrP : (Preference \times \mathbb{N}) \times Identifier \rightarrow Trace \\
TrW : (WME \times \mathbb{N}) \times Identifier \rightarrow (Preference \times \mathbb{N}) \\
learn : Learn \\
instantiation : Instantiation \\
goal : Identifier \\
results : \mathbb{P}\, Preference \\
seeds : \mathbb{P}\, Preference \\
grounds, potentials, ungrounded\_potentials : \mathbb{P}\, Condition \\
locals : \mathbb{P}(\mathrm{ran}\, PositiveCondition) \\
nots : \mathbb{P}\, Not \\
matching : Condition \rightarrow (WME \times \mathbb{N}) \\
chunking\_state : ChunkingState \\
chunking\_result : ChunkingResult \\
\hline
goal \in GM \\
\mathrm{dom}\, WM\# = WM \\
\mathrm{dom}\, PM\# = PM \\
(first \,{}_{9}^{\circ}\, second)(\!| TrP |\!) \subseteq GM \\
(first \,{}_{9}^{\circ}\, second)(\!| TrW |\!) \subseteq GM
\end{array}
$$

Chunking introduces four new memories of indefinite extent.

- *TrP* — the trace memory for preferences.
- *TrW* — the trace memory for working memory elements.
- *WM #* — a numbering function that distinguishes the instances of each working memory element in memory.
- *PM #* — a numbering function that distinguishes the instances of each preference in memory.

WM # and *PM #* are partial injections that provide a way for chunking to distinguish between two different elements that were in memory at different times. The numberings were defined in this section because only chunking must actually distinguish between the instances of memory elements. As preference phase generates new preferences, and decide and IO generate new working memory elements they extend the numbering to give each new element a new number. We chose this numbering function representation instead of giving each working memory element and preference a number component because it allowed us to still use the standard set theoretic operations, instead of having to define our own little set theory for just sets of *WMEs* and *Preferences*.

TrP records for each preference, in each goal, the trace that generated that preference instance. Similarly, *TrW* records for each working memory element instance, in each goal, the acceptable or require preference instance that was in memory when decide added the working memory element instance. The invariants on the chunking schema require that the traces are only for current goals. The specification would be more accurate if the trace memories were explicitly contracted when goals are popped, but this simpler data invariant describes the intent without making the impasser's already large operations larger.

Chunking holds a dozen other pieces of state for use during its state machine's execution.

- instantiation — the instantiation that preference phase has passed to chunking.
- goal — the match goal of the instantiation.
- results — the results of the instantiation.
- seeds — the seeds, starts out as the set of results but is deleted as each result backtraced.
- grounds, potentials, locals — the grounds, potentials and locals set of chunking's search through the trace memories.
- nots — the nots of chunking's search.

- matching — the matching that maps chunking's conditions (grounds, potentials and locals) to the working memory instances they matched.
- chunking_state — the state counter.
- chunking_result — a flag that informs the preference phase if it is receiving a new instantiation back that it should calculate support from.
- ungrounded_potentials — the set of ungrounded potentials that chunking must backtrace one step.

5.5.6 Changing, Initializing and Resetting Chunking

When chunking runs, it does not modify the contents of temporary memory. If it changed temporary memory then the calculations for the traces of instantiation from the same parallel preference phase would have ordering dependencies. We define $\Delta Chunking$ to emphasize that chunking does not modify temporary memory and that it adds to the contents of production memory.

$\Delta Chunking$

$Chunking$

$Chunking'$

$TM' = TM$

$SPM \subseteq SPM'$

InitChunking specifies legal initial states for chunking.

InitChunking

Chunking

$TrP = \emptyset$

$TrW = \emptyset$

$results = \emptyset$

$seeds = \emptyset$

$grounds = potentials = \emptyset$

$locals = \emptyset$

$nots = \emptyset$

$matching = \emptyset$

$chunking\_state = ChunkingFinishedState$

$max\_preference = 0$

$max\_wme = 0$

ChunkingInitialize initializes chunking by placing it in the initial state.

ChunkingInitialize

$\Delta Chunking$

$chunking\_state = ChunkingFinishedState$

$results = \emptyset$

$seeds = \emptyset$

$grounds = potentials = \emptyset$

$locals = \emptyset$

$nots = \emptyset$

$matching = \emptyset$

$chunking\_state' = ChunkingInitialState$

When chunking is externally reset, it must have all of its memories for the current search through trace memory emptied.

$$
\begin{array}{l}
\textit{ChunkingReset} \\
\hline
\Delta \textit{Chunking} \\
\hline
\textit{chunking\_state}' = \textit{ChunkingFinishedState} \\
\textit{results} = \emptyset \\
\textit{seeds} = \emptyset \\
\textit{grounds} = \textit{potentials} = \emptyset \\
\textit{locals} = \emptyset \\
\textit{nots} = \emptyset \\
\textit{matching} = \emptyset \\
\hline
\end{array}
$$

5.5.7 Defining Results

When an instantiation modifies the transitive closure of a goal, it is said to add results to that goal. The results are the preferences that the instantiation would make available when its right hand side's preferences are added in isolation to preference memory. These result preferences are used as the starting point, or seeds, of chunking's search through trace memory.

$$
\begin{array}{l}
\textit{Results} : \textit{Identifier} \times \textit{Instantiation} \times \mathbb{P}\ \textit{TME} \rightarrow \mathbb{P}\ \textit{Preference} \\
\hline
\forall g : \textit{Identifier};\ \imath : \textit{Instantiation};\ \textit{TM} : \mathbb{P}\ \textit{TME} \bullet \\
\quad \exists \textit{TMplusRHS} : \mathbb{P}\ \textit{TME} \quad \textit{TMplusRHS} = \textit{TM} \cup \textit{PreferenceTME}(\!| \imath.\textit{rhs} |\!) \bullet \\
\qquad \textit{Results}(g, \imath, \textit{TM}) = \\
\qquad\quad \textit{PreferenceTME}^{\sim}(\!| (\textit{OfId}_{\textit{TME}}((\textit{TCI}_{\textit{TME}}(\textit{TMplusRHS}))(\{g\}), \textit{TMplusRHS}) \\
\qquad\qquad \setminus (\textit{OfId}_{\textit{TME}}(\textit{TCI}_{\textit{TME}}(\textit{TM})(\{g\}), \textit{TM}))) \cap \operatorname{ran} \textit{PreferenceTME} |\!)
\end{array}
$$

5.5.8 Chunking an Instantiation

Chunking does one of three things with each instantiation: starts tracing it, skips it because it is in the top goal, or skips it because it generates no results.

If the instantiation has results. *ChunkingStartTracing* makes a trace for it. and initializes the search through the backtrace memories from the trace's components.

ChunkingStartTracing

Δ *Chunking*

$chunking\_state = ChunkingInitialState$

$goal' = MatchGoal(instantiation.match, TM)$

$TrP' = TrP \oplus$
$\quad \{p : instantiation.rhs \bullet$
$\quad\quad ((p, PM\#(p)), goal) \longrightarrow makeTrace(TM, instantiation)\}$

$\exists g : GM$
$\quad makeWME(goal', \text{`OBJECT'}, g, No) \in WM \wedge$
$\quad\quad Results(g, instantiation, TM) \neq \emptyset \bullet$
$\quad (results' = Results(g, instantiation, TM) \wedge$
$\quad seeds' = results')$

$grounds' = Grounds(TM, instantiation)$

$potentials' = Potentials(TM, instantiation)$

$locals' = Locals(TM, instantiation)$

$nots' = Nots(instantiation)$

$matching' = instantiation.match.matching$

$chunking\_state' = ChunkingTraceConditionState$

If the match goal of the chunk is the top goal. then it can not give any results to a parent goal. This makes it uninteresting to the chunker, so it does not need to store a trace. Chunking finishes and sets the results flag to tell preference phase that no new instantiation was generated.

ChunkingIgnoreInstantiationInTopGoal

Δ *Chunking*

$chunking\_state = ChunkingInitialState$

$\neg (\exists g : GM \bullet$
$\quad makeWME(MatchGoal(instantiation.match, TM), \text{`OBJECT'}, g, No)$
$\quad\quad \in WM)$

$chunking\_result' = NoInstantiation$

$chunking\_state' = ChunkingFinishedState$

If the trace has no results, then chunking builds the trace for its preferences and returns to preference phase.

$$\begin{array}{|l}
\textit{ChunkingNoResults} \\
\Delta \textit{Chunking} \\
\hline
\textit{chunking\_state} = \textit{ChunkingInitialState} \\
\exists g : GM \mid \textit{makeWME}(\textit{goal}, \text{`OBJECT'}, g, \textit{No}) \in WM \bullet \\
\quad \textit{Results}(g, \textit{instantiation}, TM) = \varnothing \\
\textit{TrP}' = \textit{TrP} \oplus \\
\quad \{p : \textit{instantiation.rhs} \bullet \\
\qquad ((p, PM\#(p)), \textit{goal}) \mapsto \textit{makeTrace}(TM, \textit{instantiation})\} \\
\textit{chunking\_result}' = \textit{NoInstantiation} \\
\textit{chunking\_state}' = \textit{ChunkingFinishedState}
\end{array}$$

When Decide fills out slots in working memory, it must augment the *TrW*. It gives each element a trace that maps it, under the oldest goal that it is in the transitive closure of, to the require or acceptable preference instance for it in preference memory. The *Owner* function finds the oldest goal that an identifier is in.

$$\begin{array}{|l}
\textit{Owner} : \textit{Identifier} \times \mathbb{P}\, \textit{Identifier} \times \mathbb{P}\, \textit{TME} \rightarrow \textit{Identifier} \\
\hline
\forall i : \textit{Identifier};\ GM : \mathbb{P}\, \textit{Identifier};\ TM : \mathbb{P}\, \textit{TME} \bullet \\
\qquad \exists g : GM \mid i \in \textit{TCM}(g, TM) \wedge \\
\qquad\quad \neg\, (\exists \textit{anc} : \textit{GoalAncestors}(g, TM) \bullet i \in \textit{TCM}(\textit{anc}, TM) \bullet \\
\quad \textit{Owner}(i, GM, TM) = g
\end{array}$$

TraceWME provides an operation for decide and io to use to augment the working memory instance number and trace. As preference semantics processes requires before acceptables, *TraceWME* favors requires over acceptable preferences.

TraceWME

$\Delta Chunking$
$w : WME$

$max\_wme' = max\_wme + 1$

$WM\#' = WM\# \oplus \{w \mapsto max\_wme'\}$

$\exists p : PM \mid$
$\quad p = makeUnaryPreference(first(w?).id, first(w?).attribute, first(w?).value, \text{`!'}) \bullet$
$\quad TrW' = Trw \oplus \{((w, WM\#'(w)), Owner(w.id)) \mapsto (p, PM\#(p))\}$

$\neg (\exists p : PM \bullet$
$\quad p = makeUnaryPreference(w.id, w.attribute, w.value, \text{`!'}))$
$\wedge$
$\exists p : PM \mid$
$\quad p = makeUnaryPreference(w.id, w.attribute, w.value, \text{`+'}) \bullet$
$\quad TrW' = Trw \oplus \{(w?, Owner(w.id)) \mapsto (p, PM\#(p))\}$

TraceItems adds new traces for the items augmentation of impasses. The items augmentations are traced back to the require or acceptable preference that caused them to be considered for their slot.

TraceItems

$\Delta Chunking$
$W : \mathbb{P}\ WME$
$i : Identifier$

$max\_wme' = max\_wme + \# W$

$\exists f : W \rightarrowtail (max\_wme + 1 \,..\, max\_wme') \bullet WM\#' = WM\# \oplus f$

$\exists object, attribute : Identifier \mid$
$\quad makeWME(i, \text{`OBJECT'}, object) \in WM \wedge$
$\quad makeWME(i, \text{`ATTRIBUTE'}, attribute) \in WM \bullet$
$\quad TrW' = Trw \oplus$
$\quad\quad \{w : W;\ p : PM \mid p = makeUnaryPreference(object, attribute, w.value, \text{`+'}) \bullet$
$\quad\quad\quad ((w, WM\#'(w)), Owner(w.id)) \mapsto (p, PM\#(p))\} \oplus$
$\quad\quad \{w : W;\ p : PM \mid p = makeUnaryPreference(object, attribute, w.value, \text{`!'}) \bullet$
$\quad\quad\quad ((w, WM\#'(w)), Owner(w.id)) \mapsto (p, PM\#(p))\}$

5.5.9 Tracing Seeds, Locals and Grounded Potentials

The *ChunkingTraceConditionState* performs most of the search through trace memory. Its one composite operation, *ChunkingTraceCondition*, traces through local conditions in one of seven ways.

- *ChunkingTraceSeed* — if a seed has a preference trace for the goal, then follow it.
- *ChunkingSkipSeed* — if a seed does not have a preference trace for the goal, then skip it.
- *ChunkingTraceLocal* — if a local's working memory element has a trace, then follow it.
- *ChunkingSkipArchitectureGeneratedLocal* — if a local matched an architecturally generated working memory element, skip it.
- *ChunkingTraceQuiescenceLocal* — if a local matched the goal's quiescence working memory element then add a quiescence condition to the grounds set.
- *ChunkingPotentializeLocal* — if a local condition's working memory element does not have a working memory element trace for this goal, then move it to the potential's set.
- *ChunkingTraceGroundedPotential* — if there are no more locals, and a potential's condition is in the transitive closure of the grounds set, then move it to the grounds.

The seed preferences are the starting point for chunking's search through trace memory. The seeds are processed first, and then the locals and finally the potentials. If a seed has a preference memory trace, chunking follows it by removing the seed from the seed set, and unioning the trace's components into it's search memories.

$$
\begin{array}{|l}
\hline
\textit{ChunkingTraceSeed} \\
\Delta \textit{Chunking} \\
\hline
\textit{chunking\_state} = \textit{ChunkingTraceConditionState} \doteq \textit{chunking\_state}' \\
\exists\, s : \textit{seeds} \bullet \\
\quad \exists\, t : \textit{Trace} \mid t = \textit{TrP}((s, \textit{PM}\#(s)), \textit{goal}) \bullet \\
\qquad ((\textit{seeds}' = \textit{seeds} \setminus \{s\}) \wedge \\
\qquad (\textit{grounds}' = \textit{grounds} \cup t.\textit{grounds}) \wedge \\
\qquad (\textit{potentials}' = \textit{potentials} \cup t.\textit{potentials}) \wedge \\
\qquad (\textit{locals}' = \textit{locals} \cup t.\textit{locals}) \wedge \\
\qquad (\textit{nots}' = \textit{nots} \cup t.\textit{nots}) \wedge \\
\qquad (\textit{matching}' = \textit{matching} \oplus t.\textit{matching})) \\
\hline
\end{array}
$$

Chunking skips seeds that do not have a preference trace in the match goal. They were generated in the parent goal and are in the transitive closure of the results without being part of the results.

$$
\begin{array}{l}
\textit{ChunkingSkipSeed} \\
\hline
\Delta \textit{Chunking} \\
\hline
\textit{chunking\_state} = \textit{ChunkingTraceConditionState} = \textit{chunking\_state}' \\[1ex]
\exists\, s : \textit{seeds} \mid \\
\quad \neg\,(\exists\, t : \textit{Trace} \mid t = \textit{TrP}((s, \textit{PM}\#(s)), \textit{goal})) \bullet \\
\quad\quad ((\textit{seeds}' = \textit{seeds} \setminus \{s\}) \wedge \\
\quad\quad (\textit{grounds}' = \textit{grounds}) \wedge \\
\quad\quad (\textit{potentials}' = \textit{potentials}) \wedge \\
\quad\quad (\textit{locals}' = \textit{locals}) \wedge \\
\quad\quad (\textit{nots}' = \textit{nots}) \wedge \\
\quad\quad (\textit{matching}' = \textit{matching})) \\
\hline
\end{array}
$$

Chunking's basic step in the search is to backtrace through a local. The local condition's matched working memory element is chased through the working memory trace, and the resulting preference instance is chased through the preference trace to arrive at a trace. This trace's components are unioned into the search memories, and the local is discarded.

$$
\begin{array}{l}
\textit{ChunkingTraceLocal} \\
\hline
\Delta \textit{Chunking} \\
\hline
\textit{chunking\_state} = \textit{ChunkingTraceConditionState} = \textit{chunking\_state}' \\[1ex]
\textit{seeds} = \varnothing \\[1ex]
\exists\, l : \textit{locals} \bullet \\
\quad \exists\, t : \textit{Trace} \mid \\
\quad\quad\quad t = \textit{TrP}(\textit{TrW}(\textit{matching}(l), \textit{goal}), \textit{goal}) \bullet \\
\quad\quad ((\textit{grounds}' = \textit{grounds} \cup t.\textit{grounds}) \wedge \\
\quad\quad (\textit{potentials}' = \textit{potentials} \cup t.\textit{potentials}) \wedge \\
\quad\quad (\textit{locals}' = (\textit{locals} \cup t.\textit{locals}) \setminus \{l\}) \wedge \\
\quad\quad (\textit{nots}' = \textit{nots} \cup t.\textit{nots}) \wedge \\
\quad\quad (\textit{matching}' = (\{l\} \lhd \textit{matching}) \oplus t.\textit{matching})) \\
\hline
\end{array}
$$

When decide constructs a goal or an impasse it represents it as elements in working memory. *ImpasseAttribute* is the set of the attributes it uses to represent goals and impasses.

$$
\begin{array}{|l}
\mathit{ImpasseAttribute} : \mathbb{P}\ \mathit{Constant} \\
\hline
\mathit{ImpasseAttribute} = \\
\quad \{\text{`PROBLEM-SPACE'}, \text{`STATE'}, \text{`OPERATOR'}, \text{`OBJECT'}, \\
\quad\ \text{`ATTRIBUTE'}, \text{`IMPASSE'}, \text{`CHOICES'}, \text{`ITEM'}, \text{`QUIESCENCE'} \\
\quad \text{`TYPE'}\}
\end{array}
$$

All but the items working memory elements of these goals have no working memory traces. Decide adds working memory traces that make the items depend upon the preference instance for the require or acceptable preference that caused decide to generate the impasse.

$$
\begin{array}{|l}
\hline
\textit{ChunkingSkipArchitectureGeneratedLocal} \\
\Delta \mathit{Chunking} \\
\hline
\mathit{chunking\_state} = \mathit{ChunkingTraceConditionState} = \mathit{chunking\_state}' \\
\mathit{seeds} = \varnothing \\
\exists\, l : \mathit{locals};\ w : \mathit{WME} \mid \\
\qquad w = \mathit{first}(\mathit{matching}(l)) \wedge \\
\qquad w.\mathit{id} \in \mathit{GM} \wedge \\
\qquad w.\mathit{attribute} \in \mathit{ImpasseAttribute} \setminus \{\text{`ITEM'}\} \wedge \\
\qquad w.\mathit{context\_acceptable\_preference} = \mathit{No} \bullet \\
\quad ((\mathit{grounds}' = \mathit{grounds}) \wedge \\
\quad (\mathit{potentials}' = \mathit{potentials}) \wedge \\
\quad (\mathit{locals}' = \mathit{locals} \setminus \{l\}) \wedge \\
\quad (\mathit{nots}' = \mathit{nots}) \wedge \\
\quad (\mathit{matching}' = \{l\} \lhd \mathit{matching})) \\
\hline
\end{array}
$$

The 'QUIESCENCE' goal working memory element is used to turn off chunking for a goal that the user does not want to chunk. When the user knows that the processing in a subgoal will generate a problem, she simply makes one or more of the productions that fire in the subgoal match the goal's quiescence working memory element. When chunking backtraces through this quiescence element, *ChunkingTraceQuiescenceLocal* traces it back to the quiescence element of the parent goal. When chunking has finished, the presence of the quiescence augmentation in the grounds set tells chunking not to variabilize the new chunk.

```
— ChunkingTraceQuiescenceLocal ————————————
  $\Delta Chunking$
  ————————————
  $chunking\_state = ChunkingTraceConditionState = chunking\_state'$
  $seeds = \varnothing$
  $\exists\, l : locals;\ w : WME;\ g : GM \mid$
          $w = first(matching(l)) \wedge$
          $w.id \in GM \wedge$
          $w.attribute = \text{'QUIESCENCE'} \wedge$
          $w.value = \text{'T'} \wedge$
          $w.context\_acceptable\_preference = No \wedge$
          $makeWME(goal, \text{'OBJECT'}, g, No) \in WM \bullet$
          $\exists\, qc :$
            $\{PositiveCondition($
               $makeWMETest(EqualityTest(g),$
                  $EqualityTest(\text{'QUIESCENCE'}),$
                  $EqualityTest(\text{'T'}), NoTest))\} \bullet$
            $\exists\, qw : WME \mid$
                  $qw.id = g \wedge qw.attribute = \text{'QUIESCENCE'} \wedge$
                  $qw.value = \text{'T'} \wedge qw.context\_acceptable\_preference = No \bullet$
               $((grounds' = grounds \cup \{qc\}) \wedge$
               $(potentials' = potentials) \wedge$
               $(locals' = locals \setminus \{l\}) \wedge$
               $(nots' = nots) \wedge$
               $(matching' = (\{l\} \ntriangleleft matching) \oplus \{qc \mapsto (qw, WM\#(qw))\}))$
  ————————————————————————————
```

When chunking traces back to a local that does not have an trace record for the current goal, chunking makes the local a potential. A working memory element that was accessible locally in a match but does not have trace for that goal, has to have been brought down from a parent goal. For this local to be accessible in the current goal, but not to have a trace record for the goal, it must have been brought down as part of the transitive closure of an element from a parent goal, which later had its parent goal link snapped. So chunking adds it to the set of potentials as there will be a chain of conditions through trace memory that will eventually ground out this local.

$$
\begin{array}{l}
\textit{ChunkingPotentializeLocal} \\
\Delta \textit{Chunking} \\
\hline
\textit{chunking\_state} = \textit{ChunkingTraceConditionState} = \textit{chunking\_state}' \\
\textit{seeds} = \varnothing \\
\exists\, l : \textit{locals} \mid \\
\quad \neg\, (\exists\, t : \textit{Trace} \bullet t = \textit{TrP}(\textit{TrW}(\textit{matching}(l), \textit{goal}), \textit{goal})) \bullet \\
\quad\quad ((\textit{grounds}' = \textit{grounds}) \wedge \\
\quad\quad (\textit{potentials}' = \textit{potentials} \cup \{l\}) \wedge \\
\quad\quad (\textit{locals}' = \textit{locals} \setminus \{l\}) \wedge \\
\quad\quad (\textit{nots}' = \textit{nots}) \wedge \\
\quad\quad (\textit{matching}' = \{l\} \lhd \textit{matching}))
\end{array}
$$

When there are no more seeds and locals to process, chunking adds the potentials that are connected to the grounds into the grounds set.

$$
\begin{array}{l}
\textit{ChunkingTraceGroundedPotential} \\
\Delta \textit{Chunking} \\
\hline
\textit{chunking\_state} = \textit{ChunkingTraceConditionState} = \textit{chunking\_state}' \\
\textit{seeds} = \varnothing \\
\textit{locals} = \varnothing \\
\exists\, \textit{gp} : \textit{potentials} \mid \\
\quad \textit{gp} \in \textit{TCI}_{\textit{Condition}}(\textit{grounds}, \textit{grounds} \cup \textit{potentials}) \cap \textit{potentials} \bullet \\
\quad ((\textit{grounds}' = \textit{grounds} \cup \{\textit{gp}\}) \wedge \\
\quad (\textit{potentials}' = \textit{potentials} \setminus \{\textit{gp}\}) \wedge \\
\quad (\textit{locals}' = \varnothing) \wedge \\
\quad (\textit{nots}' = \textit{nots}) \wedge \\
\quad (\textit{matching}' = \{\textit{gp}\} \lhd \textit{matching}))
\end{array}
$$

ChunkingTraceCondition is a composite operation that groups the tracing for locals and grounded potentials.

$$\begin{array}{l}
\mathit{ChunkingTraceCondition} \mathrel{\widehat{=}} \\
\quad \mathit{ChunkingTraceSeed} \lor \mathit{ChunkingSkipSeed} \lor \\
\quad \mathit{ChunkingTraceLocal} \lor \mathit{ChunkingSkipArchitectureGeneratedLocal} \lor \\
\quad \mathit{ChunkingTraceQuiescenceLocal} \lor \mathit{ChunkingPotentializeLocal} \lor \\
\quad \mathit{ChunkingTraceGroundedPotential}
\end{array}$$

5.5.10 Tracing UnGrounded Potentials

Chunking is not able to connect all of the potential conditions into the grounds set. When the seeds and locals are empty, and no potentials that are connected to the grounds set remains, *ChunkingStartTraceUnGroundedPotentials* moves chunking into the trace ungrounded potentials state. This state backtraces one step through all of the positive potentials. It saves all of the entering potentials in the *ungrounded_potentials* set, so that it can augment the potentials set with the potentials of newly backtraced traces.

$$\begin{array}{|l}
\hline
\mathit{ChunkingStartTraceUnGroundedPotentials} \\
\Delta \mathit{Chunking} \\
\hline
\mathit{chunking\_state} = \mathit{ChunkingTraceConditionState} \\
\mathit{seeds} = \varnothing \\
\mathit{locals} = \varnothing \\
\neg (\exists\, gp : \mathit{potentials} \bullet \\
\qquad gp \in \mathit{TCI}_{\mathrm{Condition}}(\mathit{grounds}, \mathit{grounds} \cup \mathit{potentials}) \cap \mathit{potentials}) \\
\mathit{potentials}' = \varnothing \\
\mathit{ungrounded\_potentials}' = \mathit{potentials} \\
\mathit{chunking\_state}' = \mathit{ChunkingTraceUnGroundedPotentialsState} \\
\hline
\end{array}$$

Each positive potential with a trace is backtraced in the standard way.

$$
\begin{array}{l}
\textit{ChunkingTraceUnGroundedPotential} \\
\hline
\Delta \textit{Chunking} \\
\hline
\textit{chunking\_state} = \textit{ChunkingTraceUnGroundedPotentialsState} = \textit{chunking\_state}' \\[1ex]
\exists\, gp : \textit{ungrounded\_potentials} \cap (\operatorname{ran} \textit{PositiveCondition}) \bullet \\
\quad \exists\, t : \textit{Trace} \mid t = \textit{TrP}(\textit{TrW}(\textit{matching}(gp), \textit{goal}), \textit{goal}) \bullet \\
\qquad ((\textit{grounds}' = \textit{grounds} \cup t.\textit{grounds}) \wedge \\
\qquad (\textit{ungrounded\_potentials}' = \textit{ungrounded\_potentials} \setminus \{gp\}) \\
\qquad (\textit{potentials}' = \cup t.\textit{potentials}) \wedge \\
\qquad (\textit{locals}' = \textit{locals} \cup t.\textit{locals}) \wedge \\
\qquad (\textit{nots}' = \textit{nots} \cup t.\textit{nots}) \wedge \\
\qquad (\textit{matching}' = \{gp\} \lhd \textit{matching})) \\
\hline
\end{array}
$$

Chunking skips positive potentials that don't have a trace.

$$
\begin{array}{l}
\textit{ChunkingSkipUnGroundedPotential} \\
\hline
\Delta \textit{Chunking} \\
\hline
\textit{chunking\_state} = \textit{ChunkingTraceUnGroundedPotentialsState} = \textit{chunking\_state}' \\[1ex]
\exists\, gp : \textit{ungrounded\_potentials} \cap (\operatorname{ran} \textit{PositiveCondition}) \mid \\
\quad \neg\, (\exists\, t : \textit{Trace} \bullet t = \textit{TrP}(\textit{TrW}(\textit{matching}(gp), \textit{goal}), \textit{goal})) \bullet \\
\qquad ((\textit{grounds}' = \textit{grounds}) \wedge \\
\qquad (\textit{ungrounded\_potentials}' = \textit{ungrounded\_potentials} \setminus \{gp\}) \\
\qquad (\textit{potentials}' = \textit{potentials}) \wedge \\
\qquad (\textit{locals}' = \textit{locals}) \wedge \\
\qquad (\textit{nots}' = \textit{nots}) \wedge \\
\qquad (\textit{matching}' = \textit{matching})) \\
\hline
\end{array}
$$

When all of the positive potentials have been traced, chunking returns to the trace condition state to backtrace the newly generated locals. The potentials collected while backtracing the positive ungrounded potentials are unioned with the remaining negated ungroundable potentials.

$$
\begin{array}{l}
\textit{ChunkingTraceUnGroundedPotentialsDone} \\
\hline
\Delta \textit{Chunking} \\
\hline
\textit{chunking\_state} = \textit{ChunkingTraceUnGroundedPotentialsState} \\[1ex]
\textit{ungrounded\_potentials} \setminus \\
\quad \{ p : \textit{ungrounded\_potentials} \mid \\
\qquad \neg\, (\exists\, t : \textit{Trace} \bullet t = \textit{TrP}(\textit{TrW}(\textit{matching}(p), \textit{goal}), \textit{goal}))\} \cap \\
\quad (\operatorname{ran} \textit{PositiveCondition}) = \varnothing \\[1ex]
\textit{potentials}' = \textit{potentials} \cup \textit{ungrounded\_potentials} \\[1ex]
\textit{ungrounded\_potentials}' = \varnothing \\[1ex]
\textit{chunking\_state}' = \textit{ChunkingTraceConditionState} \\
\hline
\end{array}
$$

5.5.11 ChunkSP

When the seeds, the locals and all of the positive potentials and the groundable negative potentials have been traced, backtracing is done and chunking generates the new production. If learning is on and the grounds do not match a quiescence test, then chunking variabilizes the grounds and results to builds the new production. Chunking also constructs the match that the matcher would generate, and uses this in constructing an instantiation for the new chunk, which it adds to the instantiation set. The new instantiation is handed back to the preference phase. Preference phase calculates the new instantiation's support, and then sends it back into chunking to have its results chunked up the hierarchy of goals.

ChunkingFinishLearnOn

$\Delta\, Chunking$

$chunking\_state = ChunkingTraceConditionState$

$seeds = \varnothing$

$locals = \varnothing$

$potentials \cap \mathrm{ran}\ PositiveCondition = \varnothing$

$learn = On$

```
$\neg\ (\exists\ qc : PositiveCondition^{\sim}\ (\!| grounds \cap \mathrm{ran}\ PositiveCondition |\!)\ \bullet$
  $EqualityTest^{\sim}(qc.id) \in GM\ \wedge$
  $EqualityTest^{\sim}(qc.attribute) \wedge =$ `QUIESCENCE' $\wedge$
    $EqualityTest^{\sim}(qc.value) =$ `T')
```

$TCI_{Condition}(grounds, grounds \cup potentials) \cap potentials = \varnothing$

```
$\exists\ a : Assignment;\ name : Symbol\ \bullet$
  $\exists\ negativeconditions : \{ VariabilizeSetOfConditions(grounds \setminus \mathrm{ran}\ PositiveCondition, a)\}\ \bullet$
    $\exists\ cwcc : \{NotifyConditionMatchings($
      $VariabilizeConditionMatching(grounds \lhd matching, a),$
        $NotsToVNots(nots, a), \varnothing)\}\ \bullet$
      $\exists\ match\_matching : \{first(first_3(cwcc))\}\ \bullet$
        $\exists\ instantiation\_matching : \{second(first_3(cwcc))\}\ \bullet$
          $\exists\ sp : SP \mid sp = makeSP(name, negativeconditions \cup \mathrm{dom}\ match\_matching,$
              $VariabilizeRHS(results, a))\ \bullet$
            $\exists\ m : Match \mid m = makeMatch(sp, match\_matching)\ \bullet$
              $\exists\ i : Instantiation \mid i = makeInstantiation(m, instantiation\_matching)\ \bullet$
                $(SPM' = SPM \cup \{sp\}\ \wedge$
                $IS' = IS \cup \{i\})$
```

$chunking\_result = NewInstantiation$

$chunking\_state' = ChunkingFinishedState$

If learning is off or quiescence is matched, chunking creates the new match and instantiation without variabilizing it and returns the instantiation to preference phase.

$$
\begin{array}{|l}
\textit{ChunkingFinishLearnOff} \\
\Delta \textit{Chunking} \\
\hline
\textit{chunking\_state} = \textit{ChunkingTraceConditionState} \\
\textit{learn} = \textit{Off} \lor \\
(\exists \textit{qc} : \textit{PositiveCondition}^{\sim} (\!| \textit{grounds} \cap \operatorname{ran} \textit{PositiveCondition} |\!) \bullet \\
\quad \textit{EqualityTest}^{\sim}(\textit{qc.id}) \in \textit{GM} \land \\
\quad \textit{EqualityTest}^{\sim}(\textit{qc.attribute}) \land = \text{`QUIESCENCE'} \land \\
\quad\quad \textit{EqualityTest}^{\sim}(\textit{qc.value}) = \text{`T'}) \\
\quad \exists \textit{m} : \textit{Match} \mid \textit{m} = \textit{makeMatch}(\textit{sp}, \textit{grounds} \lhd \textit{matching}) \bullet \\
\quad\quad \exists \textit{i} : \textit{Instantiation} \mid \textit{i} = \textit{makeInstantiation}(\textit{m}, \textit{instantiation\_matching}) \bullet \\
\quad\quad\quad (\textit{SPM}' = \textit{SPM} \cup \{\textit{sp}\} \land \\
\quad\quad\quad \textit{IS}' = \textit{IS} \cup \{\textit{i}\}) \\
\textit{chunking\_state}' = \textit{ChunkingFinishedState} \\
\hline
\end{array}
$$

5.5.12 Step Chunking

ChunkingStep groups all of the operations of chunking together into one operation that drives the entire state machine.

$$
\begin{array}{l}
\textit{ChunkingStep} \mathrel{\widehat=} \\
\quad \textit{ChunkingStartTracing} \lor \textit{ChunkingIgnoreInstantiationInTopGoal} \lor \\
\quad \textit{ChunkingNoResults} \lor \textit{ChunkingTraceCondition} \lor \\
\quad \textit{ChunkingStartTraceUnGroundedPotentials} \lor \\
\quad \textit{ChunkingTraceUnGroundedPotential} \lor \textit{ChunkingSkipUnGroundedPotential} \lor \\
\quad \textit{ChunkingTraceUnGroundedPotentialsDone} \lor \\
\quad \textit{ChunkingFinishLearnOn} \lor \textit{ChunkingFinishLearnOff}
\end{array}
$$

5.6 Slots

The modules of Soar all view preference and working memory, as being partitioned into sets of preferences for individual slots. Each slot is an identifier, symbol pair.

$$\textit{Slot} == \textit{Identifier} \times \textit{Symbol}$$

The preferences for a slot are those that hold the identifier of the slot in their id component, and the symbol of the slot in their attribute component.

Similarly, it views working memory as partitioned into sets of working memory elements for slots. The slot's working memory elements are those that hold the slot's identifier in their id component, and the symbol in their attribute, and are not context acceptable preferences.

$$
\begin{array}{l}
\mathit{SlotsWMEs} : \mathit{Slot} \times \mathbb{P}\ \mathit{WME} \rightarrow \mathbb{P}\ \mathit{WME} \\
\hline
\forall\, \mathit{slot} : \mathit{Slot};\ \mathit{WM} : \mathbb{P}\ \mathit{WME} \bullet \\
\quad \mathit{SlotsWMEs}(\mathit{slot}, \mathit{WM}) = \\
\quad\quad \{ w : \mathit{WM} \mid w.\mathit{id} = \mathit{first}(\mathit{slot}) \wedge \\
\quad\quad\quad w.\mathit{attribute} = \mathit{second}(\mathit{slot}) \wedge \\
\quad\quad\quad w.\mathit{context\_acceptable\_preference} = \mathit{No} \}
\end{array}
$$

5.7 Recognition Memory's State

Recognition memory's state schema contains a state counter for preference phase and 12 memories, some of which are shared with other modules and some of which are local to recognition memory.

The preference phase has seven states.

1. *PPhaseInitialState* — the preference phase starts execution from here.

2. *PPhaseRetractInstantiationState* — this state removes the instantiations from the instantiation set which no longer have a match that is consistent with working memory.

3. *PPhaseInstantiateMatchState* — this state instantiates matches that do not yet have an instantiation.

4. *PPhaseConferSupportState* — after a new match has been instantiated, this state calculates the right hand side preference's support, adds the support to the support memories and collects the O-supported reject preferences.

5. *PPhaseChunkingState* — preference phase steps the execution of chunking from this state.

6. *PPhaseORejectState* — any new reject preferences with O-support are processed to remove matching preferences from preference memory and the support memories.

7. *PPhaseChangePMState* — preference phase changes preference memory to match support memory.

8. *PPhaseFinishedState* — when preference phase finishes, it rests in this state.

$$
\begin{aligned}
\mathit{PreferencePhaseState} ::= \ & \mathit{PPhaseInitialState} \\
\mid \ & \mathit{PPhaseRetractInstantiationState} \\
\mid \ & \mathit{PPhaseInstantiateMatchState} \\
\mid \ & \mathit{PPhaseConferSupportState} \\
\mid \ & \mathit{PPhaseChunkingState} \\
\mid \ & \mathit{PPhaseORejectState} \\
\mid \ & \mathit{PPhaseChangePMState} \\
\mid \ & \mathit{PPhaseFinishedState}
\end{aligned}
$$

Recognition Memory's state shares six memories with other modules of Soar.

- *WM* — the set of working memory elements. *WM* is shared with IO and Decide.
- *PM* — the set of preferences. *PM* is shared with Decide.
- *TM* — essentially a union of working memory and preference memory. *TM* is shared with Decide.

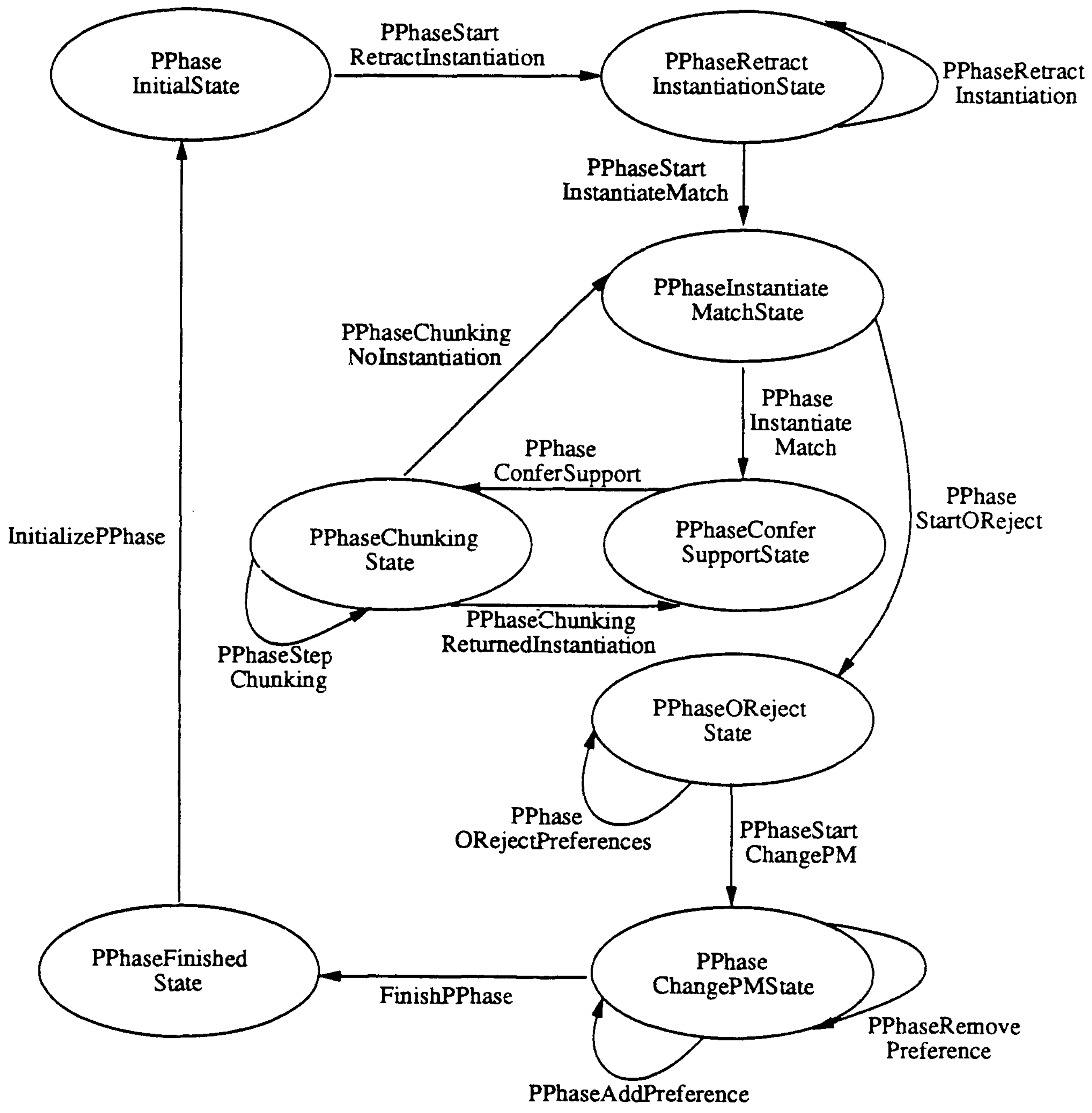

Figure 5.2: Preference Phase State Machine

- *IdentifierTable* — the set of identifiers in use in any part of Soar. The *IdentifierTable* is global to all modules of Soar.
- *GM* — is the set of goal identifiers. *GM* is read from Decide.
- *changed_slots* — the set of slots changed by preference phase. This is read and deleted by decide to tell which slots decide should consider.

RM

$WM : \mathbb{P}\ WME$

$PM : \mathbb{P}\ Preference$

$TM : \mathbb{P}\ TME$

$IdentifierTable : \mathbb{P}\ Identifier$

$GM : \mathbb{P}\ Identifier$

$SPM : \mathbb{P}\ SP$

$MS : \mathbb{P}\ Match$

$IS : \mathbb{P}\ Instantiation$

$ISM : Preference \leftrightarrow Instantiation$

$OSM : Preference \leftrightarrow Identifier$

$ORM : \mathbb{P}\ Preference$

$instantiation : Instantiation$

$pphase\_state : PreferencePhaseState$

$changed\_slots : \mathbb{P}\ Slot$

$Chunking$

$$MS = \{m : Match \mid m.production \in SPM \wedge Matches^{Match}_{WM}(m, WM, WM\#) \wedge m.binding(\!| m.production.roots |\!) \subseteq GM \cup IM\}$$

$$\operatorname{ran} OSM \subseteq GM$$

$$SPM \neq \varnothing$$

$$TM = PreferenceTME(\!| PM |\!) \cup WMETME(\!| WM |\!)$$

$$\forall p : ORM \bullet p \in \operatorname{ran} UnaryP \wedge (UnaryP^{\sim}(p)).preference = `-'$$

Recognition memory holds six local memories.

- *SPM* — is the set of productions in Soar.
- *MS* — the set of current matches for the productions. Each match must be for a production in production memory, must match the current contents of working memory, and all of the production's root variables must be bound to the identifiers of goals or impasses.
- *IS* — the set of instantiations that preference phase has created from the matches.

- *ISM* — the I-support memory. This memory is a relation from preferences to the instantiations that are currently supporting them.
- *OSM* — the O-support memory. This memory is a relation from preferences to the identifier of existing goals that had an instantiation I-support them.
- *ORM* - a memory of the O-supported reject preferences that preference phase has created while instantiating matches that have yet to be processed.

When recognition memory is initialized, all of its memories are empty except for the production memory. The production memory must not be empty so that Soar has some long term memories to use in performing its task; otherwise Soar would impasse indefinitely.

InitRM

RM

$pphase\_state = PPhaseFinishedState$

$IS = \varnothing$

$ISM = \varnothing$

$OSM = \varnothing$

$ORM = \varnothing$

$WM = \varnothing$

$PM = \varnothing$

$SPM \neq \varnothing$

5.8 Support

Support is Soar's basic mechanism for determining the extent of preferences in preference memory. Support comes in two flavors: instantiation support, I-support, and operator support, O-support. Whenever an instantiation is created, it gives instantiation support to each of the preferences in its right hand side. When the instantiation is retracted, it withdraws its instantiation support from each of its preferences. Whenever instantiations that are creating, modifying or applying operators create a preference whose identifier is in the transitive closure of their match goal's states or operators, they confer operator support to these preference. The operator support has the extent of the match goal of the instantiation, and so is not retracted when the instantiation retracts.

This section defines the operators matched by an instantiation, when an instantiation is creating, modifying or applying an operator, and then combines these into a definition of when an instantiation O-supports a preference.

An instantiation matches an operator if it matches a goal's operator slot or operator acceptable preference.

$$\begin{array}{l}
\mathit{InstantiationsLHSOperators} : \mathit{Instantiation} \times \mathbb{P}\ \mathit{TME} \rightarrow \mathbb{P}\ \mathit{Identifier} \\
\hline
\forall\, i : \mathit{Instantiation};\ \mathit{TM} : \mathbb{P}\ \mathit{TME} \bullet \\
\quad \exists\, g : \mathit{Identifier} \mid g = \mathit{MatchGoal}(i.\mathit{match}, \mathit{TM}) \bullet \\
\quad\quad \mathit{InstantiationsLHSOperators}(i, \mathit{TM}) = \\
\quad\quad\quad \{o : \mathit{Identifier} \mid \mathit{makeWME}(g, \text{`OPERATOR'}, o, \mathit{No}) \in i.\mathit{match}.\mathit{lhs} \lor \\
\quad\quad\quad\quad \mathit{makeWME}(g, \text{`OPERATOR'}, o, \mathit{Yes}) \in i.\mathit{match}.\mathit{lhs}\}
\end{array}$$

An instantiation creates an operator if it makes an acceptable or require preference for an operator slot of a goal.

$$\begin{array}{l}
\mathit{InstantiationsRHSOperators} : \mathit{Instantiation} \times \mathbb{P}\ \mathit{TME} \rightarrow \mathbb{P}\ \mathit{Identifier} \\
\hline
\forall\, i : \mathit{Instantiation};\ \mathit{TM} : \mathbb{P}\ \mathit{TME} \bullet \\
\quad \mathit{InstantiationsRHSOperators}(i, \mathit{TM}) = \\
\quad\quad \{o : \mathit{Identifier} \mid \\
\quad\quad\quad \mathit{makeUnaryPreference}(\mathit{MatchGoal}(i.\mathit{match}, \mathit{TM}), \\
\quad\quad\quad\quad \text{`OPERATOR'}, o, \text{`}-\text{'}) \in i.\mathit{rhs} \lor \\
\quad\quad\quad \mathit{makeUnaryPreference}(\mathit{MatchGoal}(i.\mathit{match}, \mathit{TM}), \\
\quad\quad\quad\quad \text{`OPERATOR'}, o, \text{`!'}) \in i.\mathit{rhs}\}
\end{array}$$

An instantiation only confers operator creation support if its match matched a working memory element whose identifier is in the transitive closure of the match goal's state.

$$\begin{array}{l}
\mathit{OperatorCreationLHS} : \mathbb{P}\ \mathit{TME} \rightarrow \mathbb{P}\ \mathit{Instantiation} \\
\hline
\forall\, \mathit{TM} : \mathbb{P}\ \mathit{TME};\ i : \mathit{Instantiation} \bullet \\
\quad i \in \mathit{OperatorCreationLHS}(\mathit{TM}) \Leftrightarrow \\
\quad\quad (\exists\, g : \{\mathit{MatchGoal}(i.\mathit{match}, \mathit{TM})\} \bullet \\
\quad\quad\quad (\exists\, s : \{\mathit{GoalState}(g, \mathit{TM})\} \bullet \\
\quad\quad\quad\quad (\mathit{OfId}_{\mathit{TME}}((\mathit{TCI}_{\mathit{TME}}(\mathit{TM}))(\{s\}), \mathit{TM}) \cap \\
\quad\quad\quad\quad\quad \mathit{WMETME}(\!| i.\mathit{match}.\mathit{lhs} |\!) \neq \emptyset)))
\end{array}$$

An instantiation's right hand side only confers operator support on a preference if the preference's identifier is in the transitive closure of an instantiation's operator over temporary memory and the instantiation's right hand side.

$$
\begin{array}{|l}
OperatorCreationRHS : \mathbb{P}\ TME \rightarrow (Instantiation \leftrightarrow Preference) \\
\hline
\forall\ TM : \mathbb{P}\ TME;\ i : Instantiation;\ p : Preference \bullet \\
\quad ((i, p) \in OperatorCreationRHS(TM)) \Leftrightarrow \\
\quad\quad (PreferencesId(p) \in \\
\quad\quad\quad (TCI_{TME}(TM \cup PreferenceTME(\!|i.rhs|\!))) \\
\quad\quad\quad\quad (InstantiationsRHSOperators(i, TM)))
\end{array}
$$

An instantiation confers operator creation support on a preference if its left hand side is an operator creation and the right hand side confers operator creation support on the preference.

$$
\begin{array}{|l}
OperatorCreation : \mathbb{P}\ TME \rightarrow (Instantiation \leftrightarrow Preference) \\
\hline
\forall\ TM : \mathbb{P}\ TME;\ i : Instantiation;\ p : Preference \bullet \\
\quad (i, p) \in OperatorCreation(TM) \\
\quad\quad \Leftrightarrow i \in OperatorCreationLHS(TM) \wedge \\
\quad\quad\quad (i, p) \in OperatorCreationRHS(TM)
\end{array}
$$

An instantiation can confer operator modification support if its left hand side matches an operator and its match matches a working memory element of the state.

$$
\begin{array}{|l}
OperatorModificationLHS : \mathbb{P}\ TME \rightarrow \mathbb{P}\ Instantiation \\
\hline
\forall\ TM : \mathbb{P}\ TME;\ i : Instantiation \bullet \\
\quad i \in OperatorModificationLHS(TM) \Leftrightarrow \\
\quad\quad (\exists\ g : \{MatchGoal(i.match, TM)\} \bullet \\
\quad\quad\quad (\exists\ s : \{GoalState(g, TM)\} \bullet \\
\quad\quad\quad\quad ((OfId_{TME}(TCI_{TME}(TM)(\{s\}), TM) \cap \\
\quad\quad\quad\quad\quad WMETME(\!|i.match.lhs|\!)) \neq \emptyset) \wedge \\
\quad\quad\quad\quad\quad (InstantiationsLHSOperators(i, TM) \neq \emptyset)))
\end{array}
$$

An instantiation's right hand side confers operator modification support on a preference if its identifier is in the transitive closure of a matched operator.

$$
\begin{array}{|l}
\mathit{OperatorModificationRHS} : \mathbb{P}\ \mathit{TME} \rightarrow (\mathit{Instantiation} \leftrightarrow \mathit{Preference}) \\
\hline
\forall\ \mathit{TM} : \mathbb{P}\ \mathit{TME};\ i : \mathit{Instantiation};\ p : \mathit{Preference} \bullet \\
\quad (i, p) \in \mathit{OperatorModificationRHS}(\mathit{TM}) \Leftrightarrow \\
\quad\quad (\exists\ g : \{\mathit{MatchGoal}(i.match, \mathit{TM})\} \bullet \\
\quad\quad\quad \mathit{PreferencesId}(p) \in \\
\quad\quad\quad\quad (\mathit{TCI}_{\mathit{TME}}(\mathit{TM} \cup \mathit{PreferenceTME}(\!| i.rhs |\!))) \\
\quad\quad\quad\quad\quad (\mathit{InstantiationsLHSOperators}(i, \mathit{TM})))
\end{array}
$$

An instantiation confers operator modification support on a preference if the left hand side is an operator modification left hand side and the right hand side confers operator modification support on the preference.

$$
\begin{array}{|l}
\mathit{OperatorModification} : \mathbb{P}\ \mathit{TME} \rightarrow (\mathit{Instantiation} \leftrightarrow \mathit{Preference}) \\
\hline
\forall\ \mathit{TM} : \mathbb{P}\ \mathit{TME};\ i : \mathit{Instantiation};\ p : \mathit{Preference} \bullet \\
\quad (i, p) \in \mathit{OperatorModification}(\mathit{TM}) \Leftrightarrow \\
\quad\quad i \in \mathit{OperatorModificationLHS}(\mathit{TM}) \wedge \\
\quad\quad (i, p) \in \mathit{OperatorModificationRHS}(\mathit{TM})
\end{array}
$$

An instantiation can confer operator application support if its left hand side matches its match goal's operator slot and a working memory element in the transitive closure of the match goal's state.

$$
\begin{array}{|l}
\mathit{OperatorApplicationLHS} : \mathbb{P}\ \mathit{TME} \rightarrow \mathbb{P}\ \mathit{Instantiation} \\
\hline
\forall\ \mathit{TM} : \mathbb{P}\ \mathit{TME};\ i : \mathit{Instantiation} \bullet \\
\quad i \in \mathit{OperatorApplicationLHS}(\mathit{TM}) \Leftrightarrow \\
\quad\quad (\exists\ g : \{\mathit{MatchGoal}(i.match, \mathit{TM})\} \bullet \\
\quad\quad\quad (\exists\ s : \{\mathit{GoalState}(g, \mathit{TM})\} \bullet \\
\quad\quad\quad\quad (\exists\ o : \{\mathit{GoalOperator}(g, \mathit{TM})\} \bullet \\
\quad\quad\quad\quad\quad \mathit{makeWME}(g, \text{`OPERATOR'}, o, \mathit{No}) \in i.match.lhs \wedge \\
\quad\quad\quad\quad\quad (\mathit{OfId}_{\mathit{TME}}((\mathit{TCI}_{\mathit{TME}}(\mathit{TM}))(\{s\}), \mathit{TM}) \cap \\
\quad\quad\quad\quad\quad \mathit{WMETME}(\!| i.match.lhs |\!) \neq \emptyset))))
\end{array}
$$

An instantiation can confer operator application support on a preference if the preference's identifier is in the transitive closure of the state.

$$
\begin{array}{l}
\mathit{OperatorApplicationRHS} : \mathbb{P}\ \mathit{TME} \rightarrow (\mathit{Instantiation} \leftrightarrow \mathit{Preference}) \\
\hline
\forall\ \mathit{TM} : \mathbb{P}\ \mathit{TME};\ i : \mathit{Instantiation};\ p : \mathit{Preference} \bullet \\
\quad (i, p) \in \mathit{OperatorApplicationRHS}(\mathit{TM}) \Leftrightarrow \\
\quad\quad (\exists\ g : \{\mathit{MatchGoal}(i.\mathit{match}, \mathit{TM})\} \bullet \\
\quad\quad\quad (\exists\ s : \{\mathit{GoalState}(g, \mathit{TM})\} \bullet \\
\quad\quad\quad\quad\quad \mathit{PreferencesId}(p) \in \mathit{TCI}_{\mathit{TME}}(\mathit{TM} \cup \mathit{PreferenceTME}(\!| i.\mathit{rhs} |\!))(\{s\}))))
\end{array}
$$

An instantiation confers operator application support on a preference if the left hand side is an operator application left hand side and the right hand side confers operator application support on the preference.

$$
\begin{array}{l}
\mathit{OperatorApplication} : \mathbb{P}\ \mathit{TME} \rightarrow (\mathit{Instantiation} \leftrightarrow \mathit{Preference}) \\
\hline
\forall\ \mathit{TM} : \mathbb{P}\ \mathit{TME};\ i : \mathit{Instantiation};\ p : \mathit{Preference} \bullet \\
\quad (i, p) \in \mathit{OperatorApplication}(\mathit{TM}) \Leftrightarrow \\
\quad\quad i \in \mathit{OperatorApplicationLHS}(\mathit{TM}) \wedge \\
\quad\quad (i, p) \in \mathit{OperatorApplicationRHS}(\mathit{TM})
\end{array}
$$

An instantiation confers operator support on a preference if the instantiation confers operator creation, modification or application support on the preference.

$$
\begin{array}{l}
\mathit{OSupport} : \mathbb{P}\ \mathit{TME} \rightarrow (\mathit{Instantiation} \leftrightarrow \mathit{Preference}) \\
\hline
\forall\ \mathit{TM} : \mathbb{P}\ \mathit{TME};\ i : \mathit{Instantiation};\ p : \mathit{Preference} \bullet \\
\quad (i, p) \in \mathit{OSupport}(\mathit{TM}) \Leftrightarrow \\
\quad\quad (i, p) \in (\mathit{OperatorCreation}(\mathit{TM}) \cup \\
\quad\quad\quad\quad \mathit{OperatorModification}(\mathit{TM}) \cup \\
\quad\quad\quad\quad \mathit{OperatorApplication}(\mathit{TM}))
\end{array}
$$

5.9 Preference Phase Operations

The preference phase has eight states: an initial state, a final state, a state for retracting instantiations, a state for instantiating matches, a state for conferring support, a state for chunking an instantiation, a state for O-rejecting preferences and a state for changing the contents of preference memory based upon the contents of the support memories. This section specifies the operations of these states in seven sections.

5.9.1 Initialize, Finish, Reset and Quiescence

The preference phase waits in its final state until it is initialized with the *PPhaseInitialize* operation.

PPhaseInitialize

ΔRM

$pphase\_state = PPhaseFinishedState$

$pphase\_state' = PPhaseInitialState$

The *PPhaseReset* operation resets the preference phase state machine. The set of waiting O-reject preferences is emptied so that the next execution of the preference phase does not reject things that no longer have O-supported rejects.

PPhaseReset

ΔRM

$pphase\_state' = PPhaseFinishedState$

$ORM' = \emptyset$

The top level state machine checks the state predicate, *Quiescence*, before it allows the preference phase to run. The system is quiescent if there is no instantiation waiting to be retracted, or match waiting to be instantiated.

Quiescence

RM

$\neg (\exists i : IS \bullet i.match \notin MS)$

$\neg (\exists m : MS \bullet \neg (\exists i : IS \bullet i.match = m))$

5.9.2 Retracting Instantiations

Preference phase begins its execution by moving from the initial state into a state that retracts instantiations.

PPhaseStartRetractInstantiation

ΔRM

$pphase\_state = PPhaseInitialState$

$pphase\_state' = PPhaseRetractInstantiationState$

When there exists an instantiation in instantiation memory whose match is no longer in the match set, then *PPhaseRetractInstantiation* removes the instantiation from the instantiation set. The instantiation support for each preference of the right hand side is removed from I-support memory.

PPhaseRetractInstantiation

ΔRM

$pphase\_state = PPhaseRetractInstantiationState = pphase\_state'$

$$\exists\, i : IS \mid i.match \notin MS \bullet$$
$$(IS' = IS \setminus \{i\} \wedge$$
$$ISM' = ISM \setminus \{p : i.rhs \bullet (p \mapsto i)\})$$

5.9.3 Instantiate Match

When there is no instantiation in the instantiation set whose match is no longer satisfied, preference phase moves into the instantiate match state.

PPhaseStartInstantiateMatch

ΔRM

$pphase\_state = PPhaseRetractInstantiationState$

$\neg\, (\exists\, i : IS \bullet i.match \notin MS)$

$pphase\_state' = PPhaseInstantiateMatchState$

PPhaseInstantiateMatch selects a match from the match set that does not yet have an instantiation and instantiates it. An instantiation is created and added to the instantiation memory, and recognition memory's *instantiation* variable is set to this new instantiation. Each newly generated preference is given a new instance numbering in $PM\#$

$$
\begin{array}{l}
\textit{PPhaseInstantiateMatch} \\
\hline
\Delta RM \\
\hline
pphase\_state = PPhaseInstantiateMatchState \\[1ex]
\exists\, m : MS \mid \neg\, (\exists\, i : IS \bullet i.match = m) \bullet \\
\quad (\exists\, i : Instantiation;\ b : Binding \mid \\
\qquad i.match = m \land \\
\qquad i.lhs = m.production.lhs \land \\
\qquad i.matching = \\
\qquad\quad InstantiateConditionMatching(m.matching, \\
\qquad\qquad m.production.lhs \setminus \operatorname{ran} PositiveCondition, m.binding) \land \\
\qquad b\ ConsistentExtension\ m.binding \land \\
\qquad b\ Covers\ (\bigcup (MakesComponents\, (\!| m.production.rhs |\!)) \cap Variable) \land \\
\qquad\quad i.binding = b \land \\
\qquad i.rhs = (MakesPreference(b))\, (\!| m.production.rhs |\!) \bullet \\
\quad\quad instantiation' = i \land \\
\quad\quad IS' = IS \cup \{i\} \land \\
\quad\quad \exists\, new\_preferences : \mathbb{P}\ Preference \mid new\_preferences = (i.rhs \setminus PM) \bullet \\
\qquad max\_preference' = max\_preference + \#new\_preferences \bullet \\
\qquad\quad \exists\, f : new\_preferences \rightarrowtail ((max\_preference + 1) \,..\, max\_preference') \bullet \\
\qquad\quad PM\#' = PM\# \oplus f) \\[1ex]
pphase\_state' = PPhaseConferSupportState \\
\hline
\end{array}
$$

5.9.4 Conferring Support

The *PPhaseConferSupport* operation adds support to the I-support memory for each right hand side preference, and if the preference is O-supported then it is given O-support for the match goal. Each reject preference that has O-support is added to the memory of O-reject preferences. After *PPhaseConferSupport* operates, chunking is stepped for the new instantiation.

```
PPhaseConferSupport
  ΔRM
  ─────────
  pphase_state = PPhaseConferSupportState
      ISM' = ISM ∪ {p : instantiation.rhs • (p, instantiation)} ∧
      OSM' = OSM ∪
        {p : instantiation.rhs |
          (instantiation, p) ∈ OSupport(TM) •
      (p, MatchGoal(instantiation.match, TM))} ∧
      ORM' = ORM ∪
        {p : instantiation.rhs ∩ ran UnaryP |
          (UnaryP~(p)).preference = '−' ∧ (instantiation, p) ∈ OSupport(TM)}
  ChunkingInitialize
  pphase_state' = PPhaseChunkingState
```

5.9.5 Stepping Chunking

PPhaseStepChunking steps the chunking state machine through chunking the instantiation.

```
PPhaseStepChunking
  ΔRM
  ─────────
  pphase_state = PPhaseChunkingState = pphase_state'
  ChunkingStep
```

Chunking may return a new instantiation or it may produce no new instantiation if the input instantiation is for the top goal. The variable *chunking_result* signals the existence of a new instantiation to preference phase.

If there is a new instantiation, it must have its support calculated, so preference phase returns to the confer support state. Once the support has been conferred, the new instantiation will be sent back into chunking so that its effects may be chunked.

$$
\begin{array}{|l}
\hline
\textit{PPhaseChunkingReturnedInstantiation} \\
\Delta RM \\
\hline
pphase\_state = PPhaseChunkingState \\
\neg \ \mathrm{pre}\ Chunking \\
chunking\_result = NewInstantiation \\
pphase\_state' = PPhaseConferSupportState \\
\hline
\end{array}
$$

If no instantiation has been generated by chunking, the preference phase returns to the instantiate match state to pick a new match for instantiation.

$$
\begin{array}{|l}
\hline
\textit{PPhaseChunkingNoInstantiation} \\
\Delta RM \\
\hline
pphase\_state = PPhaseChunkingState \\
\neg \ \mathrm{pre}\ Chunking \\
chunking\_result = NoInstantiation \\
pphase\_state' = PPhaseInstantiateMatchState \\
\hline
\end{array}
$$

5.9.6 O-rejects

When there are no more matches to instantiate, preference phase starts the processing of O-rejects.

$$
\begin{array}{|l}
\hline
\textit{PPhaseStartOReject} \\
\Delta RM \\
\hline
pphase\_state = PPhaseInstantiateMatchState \\
\forall\, m : MS \bullet \exists\, i : IS \bullet i.match = m \\
pphase\_state' = PPhaseORejectState \\
\hline
\end{array}
$$

PPhaseORejectPreferences chooses an O-reject preference from the O-reject memory and deletes it from memory. It also deletes from preference memory all preferences that share the same identifier, attribute and value. The I-support and O-support memories are also cleaned of all support for the rejected preferences, including the O-reject itself. The ⩤ operation ([Spi89] 99) restricts a relation or partial function to apply only to those elements of its domain that are not in the provided set.

$$
\begin{array}{l}
\textit{PPhaseORejectPreferences} \\
\hline
\Delta RM \\
\hline
pphase\_state = PPhaseORejectState = pphase\_state' \\[4pt]
\exists\, r : ORM;\ ur : UnaryPreference \mid ur = UnaryP^{\sim}(r) \bullet \\
\quad (ORM' = ORM \setminus \{r\} \land \\
\quad (\exists\, P : \mathbb{P}\, Preference \mid \\
\qquad P = UnaryP(\!|\{up : UnaryP^{\sim}(\!|((PM \cup \operatorname{dom} ISM) \cap \operatorname{ran} UnaryP)|\!) \mid \\
\qquad\quad up.id = ur.id \land up.attribute = ur.attribute \land up.value = ur.value\}|\!) \cup \\
\qquad\quad BinaryP(\!|\{bp : BinaryP^{\sim}(\!|((PM \cup \operatorname{dom} ISM) \cap \operatorname{ran} BinaryP)|\!) \mid \\
\qquad\quad bp.id = ur.id \land bp.attribute = ur.attribute \land bp.value = ur.value\}|\!) \cup \\
\qquad\quad \{r\} \bullet \\
\qquad PM' = PM \setminus P \land \\
\qquad ISM' = P \mathbin{⩤} ISM \land \\
\qquad OSM' = P \mathbin{⩤} OSM)) \\
\hline
\end{array}
$$

5.9.7 Changing Preference Memory

When all of the O-rejects have been processed, preference phase starts changing preference memory to correspond with the contents of the support memories.

$$
\begin{array}{l}
\textit{PPhaseStartChangePM} \\
\hline
\Delta RM \\
\hline
pphase\_state = PPhaseORejectState \\[4pt]
ORM = \varnothing \\[4pt]
pphase\_state' = PPhaseChangePMState \\
\hline
\end{array}
$$

If there is a preference that has support in I-support memory or O-support memory that is not in preference memory, then it is added to preference memory, and its slot is added to the set of changed slots.

PPhaseAddPreference

ΔRM

$$pphase\_state = PPhaseChangePMState = pphase\_state'$$

$$\begin{aligned}&\exists p : \operatorname{dom} ISM \cup \operatorname{dom} OSM \mid p \notin PM \bullet \\ &\quad (PM' = PM \cup \{p\} \land \\ &\quad\ changed\_slots' = changed\_slots \cup \{(p.id, p.attribute)\})\end{aligned}$$

If there is a preference in preference memory that does not have I-support or O-support, it is removed from preference memory, and its slot is added to the changed slots.

PPhaseRemovePreference

ΔRM

$$pphase\_state = PPhaseChangePMState = pphase\_state'$$

$$\begin{aligned}&\exists p : PM \mid p \notin \operatorname{dom} ISM \cup \operatorname{dom} OSM \bullet \\ &\quad (PM' = PM \setminus \{p\} \land \\ &\quad\ changed\_slots' = changed\_slots \cup \{(p.id, p.attribute)\}))\end{aligned}$$

When there are no more differences between the supported preferences and the contents of preference memory, preference phase is complete.

PPhaseFinish

ΔRM

$$pphase\_state = PPhaseChangePMState$$

$$\neg (\exists p : PM \bullet p \notin \operatorname{dom} ISM \cup \operatorname{dom} OSM)$$

$$\neg (\exists p : \operatorname{dom} ISM \bullet p \notin PM)$$

$$pphase\_state' = PPhaseFinishedState$$

PPhaseStep moves the preference phase state machine through one of its possible transitions.

$$
\begin{aligned}
&\mathit{PPhaseStep} \mathrel{\widehat{=}} \\
&\quad \mathit{PPhaseStartRetractInstantiation} \lor \mathit{PPhaseRetractInstantiation} \lor \\
&\quad \mathit{PPhaseStartInstantiateMatch} \lor \mathit{PPhaseInstantiateMatch} \lor \\
&\quad \mathit{PPhaseStartOReject} \lor \mathit{PPhaseORejectPreferences} \lor \\
&\quad \mathit{PPhaseStartChangePM} \lor \mathit{PPhaseAddPreference} \lor \\
&\quad \mathit{PPhaseRemovePreference} \lor \mathit{PPhaseFinish}
\end{aligned}
$$

Chapter 6

IO

This chapter specifies Soar's mechanism for communicating with the external world, called SoarIO or IO for short. IO's operation is graphically represented in Figure 6.1. IO's two state machines, *InputCycle* and *OutputCycle*, are stepped by the top level state machine. The *InputCycle* reads *InputStructures* from *InputChannels* and places their results on *InputAttributes* on the top state, and in the transitive closure of the value of the input attribute. The *OutputCycle* packages the transitive closures of output attributes of the top state into *OutputStructures* and ships them out *OutputChannels*.

6.1 Channels, Attributes and IO Mappings

The user of Soar provides IO with two fixed sets: *InputChannel* and *OutputChannel*. For the purposes of the specification we represent these channels as a sets of constant names for the channels. The implementation may choose to represent a channel as a function name, or a pointer to a function; in essence it is a computational operation that provides input perceptions or receives motor operations.

The user also specifies two sets of *InputAttributes* and *OutputAttributes*. The *InputAttributes* are the attributes that input channels use to augment the top level state with their perceptions. The user provided *InputMapping* maps each *InputChannel* to the *InputAttribute* that receives its perceptions. Similarly, the *OutputAttributes* are the attributes of the top level state from which IO collects their transitive closure and ships it to the *OutputChannel* specified by the *OutputMapping*.

No channel can be both an input and an output channel, and no attribute can be both an input and an output attribute.

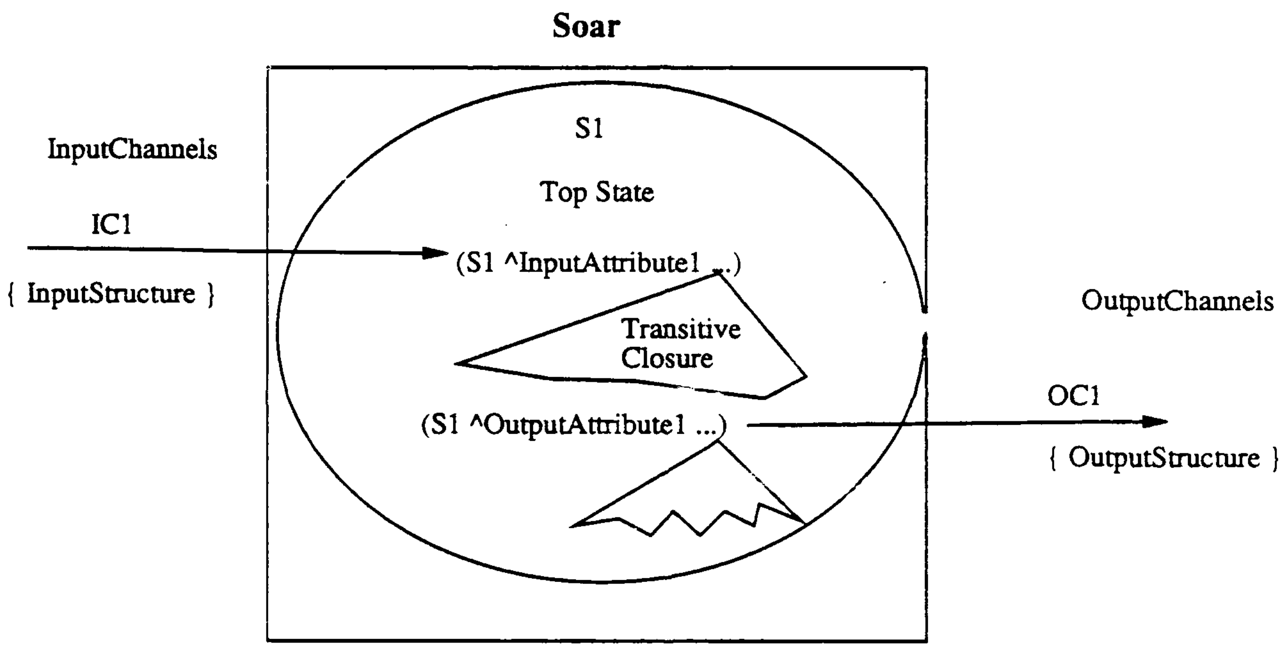

Figure 6.1: Graphical Representation of IO

$InputChannel : \mathbb{P}\ Constant$
$InputAttribute : \mathbb{P}\ Constant$
$OutputChannel : \mathbb{P}\ Constant$
$OutputAttribute : \mathbb{P}\ Constant$

disjoint $\langle InputChannel, OutputChannel \rangle$

disjoint $\langle InputAttribute, OutputAttribute \rangle$

$InputMapping : InputChannel \rightarrow InputAttribute$
$OutputMapping : OutputAttribute \twoheadrightarrow OutputChannel$

6.2 Output Structures

The *OutputCycle* collects output into packages called *OutputStructures*. On each cycle all *OutputAttributes* have the working memory elements in their transitive closure collected into an *OutputStructure* and shipped out their corresponding *OutputChannel*. The predicate of the output structure ensures that the set of working memory elements is transitively closed from the object identifier.

OutputStructure
$object : Identifier$
$wmes : \mathbb{P}\ WME$

$OfId_{WME}((TCI_{WME}(wmes))(\{object\}), wmes) = wmes$

6.3 Input Structures

On each *InputCycle*, all of the *InputChannels* are polled to produce changes to working memory. These changes are bundled into *InputStructures*. There are three types of input structures: a *NewInputStructure*, a *ModifyInputStructure* and a *DeleteInputStructure*.

A *NewInputStructure* tells Soar to augment the top state with a working memory of value *new_object* and the attribute of the input channel, and to also add all of the elements in *wmes*.

NewInputStructure
$new\_object : Identifier$
$wmes : \mathbb{P}\ WME$

$OfId_{WME}((TCI_{WME}(wmes))(\{new\_object\}), wmes) = wmes$

A *ModifyInputStructure* tells Soar to find the augmentation of the top state that holds the input channel's attribute and the *modify_object* value. Soar must add all of the elements in the *adds* set, and delete all of the elements in the *deletes* set.

$$
\begin{array}{|l}
\hline
\textit{ModifyInputStructure} \\
\textit{modify\_object} : \textit{Identifier} \\
\textit{adds}, \textit{deletes} : \mathbb{P}\ \textit{WME} \\
\hline
\end{array}
$$

The *DeleteInputStructure* tells Soar to delete the augmentation of the input channel's attribute and the *delete_object* value and all of the elements in its transitive closure.

$$
\begin{array}{|l}
\hline
\textit{DeleteInputStructure} \\
\textit{delete\_object} : \textit{Identifier} \\
\hline
\end{array}
$$

The free type *InputStructure* is used to construct a disjunctive type of the basic input types.

$$
\begin{array}{rl}
\textit{InputStructure} ::= & \textit{New}\langle\langle \textit{NewInputStructure} \rangle\rangle \\
\mid & \textit{Modify}\langle\langle \textit{ModifyInputStructure} \rangle\rangle \\
\mid & \textit{Delete}\langle\langle \textit{DeleteInputStructure} \rangle\rangle
\end{array}
$$

6.4 Cycles of IO

As Soar behaves, it reads in parallel from all of its input channels. Each input channel is free to send more than one *InputStructure* to the system. A CycleOfInput is defined to capture the input for a single cycle. It maps each *InputChannel* to the set of inputs read during that cycle.

$$
\begin{array}{l}
\textit{CycleOfInput} == \textit{InputChannel} \rightarrow \mathbb{P}\ \textit{InputStructure} \\
\textit{CycleOfOutput} == \textit{OutputChannel} \rightarrow \textit{OutputStructure}
\end{array}
$$

Similarly, each parallel ply of Soar's motor actions are captured in a partial function from *OutputChannels* to the single possible *OutputStructure* that Soar produced.

6.5 IO State

IO requires two state machines: one for the *InputCycle* (Figure 6.2) and one for the *OutputCycle* (Figure 6.3).

The *InputCycle* requires three states.

1. InputCycleInitialState — the initial state of the machine
2. InputCycleReadState — the state in which a single *InputStructure* is read from an *InputChannel*
3. InputCycleFinishedState — the final state of the machine

$$\begin{aligned} InputCycleState ::= & \; InputCycleInitialState \\ | & \; InputCycleReadState \\ | & \; InputCycleFinishedState \end{aligned}$$

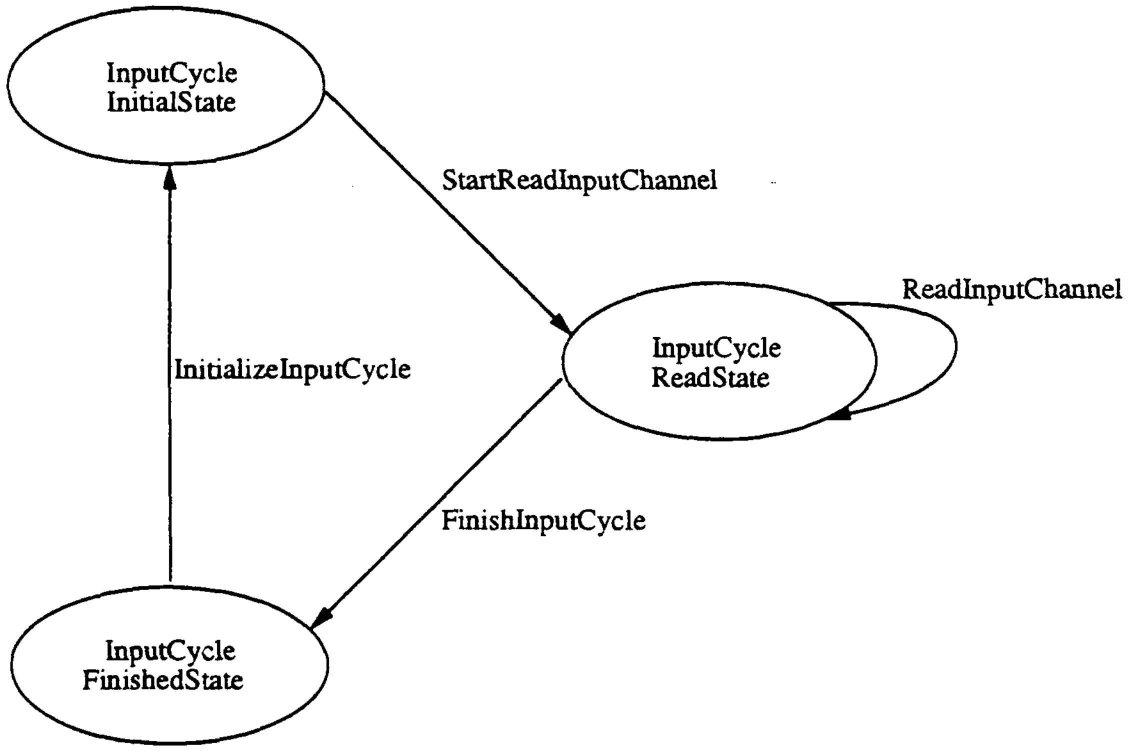

Figure 6.2: Input Cycle State Machine

The *OutputCycle* also requires only three states:

1. InputCycleInitialState — the initial state of the machine.
2. InputCycleReadState — the state in which an *OutputStructure* is collected from the top level state and sent out an *OutputChannel*.
3. InputCycleFinishedState — the final state of the machine.

OutputCycleState ::= *OutputCycleInitialState*
| *OutputCycleSendState*
| *OutputCycleFinishedState*

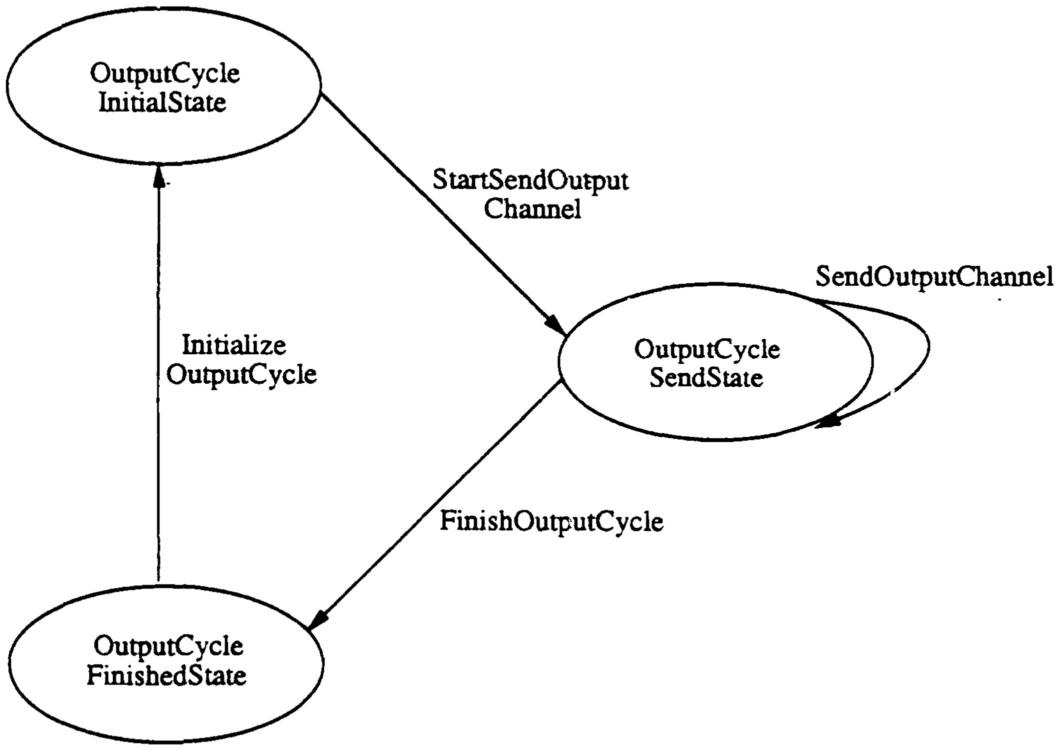

Figure 6.3: Output Cycle State Machine

The *IO* state schema shares working memory and goal memory with the other major modules of Soar. The IO system searches working memory to keep updated the top state of the goal stack. If there is no top state, then *TopState* is set to a special symbol 'NIL'.

The schema holds state counters *input_cycle_state* and *output_cycle_state* for the two IO state machines.

The specification of *IO* takes a traditional approach to modeling input/output behavior by using sequences of inputs to model the sequence of inputs that Soar would encounter over time, and by recording its outputs over time in a sequence.

inputs is the sequence of *CycleOfInput* that models a possibly infinite stream of cycles of input that Soar will perceive as it behaves. *outputs* is the initially empty sequence of *CycleOfOutput* in which IO records Soar's output over time.

Other approaches to modeling the input/output behavior of a computational system are possible. For example, the operations that read input from the sequence of inputs can be replaced by an under-specified operation that produces any one input from the set of all possible inputs. Similarly, the output operation can be under-specified to have no effect.

$$
\begin{array}{l}
\textit{IO} \\
\hline
WM : \mathbb{P}\ WME \\
GM : \mathbb{P}\ Identifier \\
TopState : Identifier \cup \{\text{`NIL'}\} \\
inputs : \text{seq}\ CycleOfInput \\
input\_cycle\_state : InputCycleState \\
outputs : \text{seq}\ CycleOfOutput \\
output\_cycle\_state : OutputCycleState \\
\hline
\exists g : GM \mid makeWME(g, \text{`OBJECT'}, \text{`NIL'}) \in WM \bullet \\
\quad ((\exists s : Identifier \bullet makeWME(g, \text{`STATE'}, s) \in WM) \Rightarrow \\
\quad\quad (\exists s : Identifier \mid makeWME(g, \text{`STATE'}, s) \in WM \bullet \\
\quad\quad\quad TopState = S) \\
\quad\quad \neg (\exists s : Identifier \bullet makeWME(g, \text{`STATE'}, s) \in WM) \Rightarrow \\
\quad\quad\quad TopState = \text{`NIL'}) \\
\hline
\end{array}
$$

IO starts with both state machines in their finished state and an empty sequence of outputs.

$$
\begin{array}{l}
\textit{InitIO} \\
\hline
IO \\
\hline
input\_cycle\_state = InputCycleFinishedState \\
output\_cycle\_state = OutputCycleFinishedState \\
outputs = \langle\rangle \\
\hline
\end{array}
$$

6.6 The Input Cycle

The *InputCycle* is defined using five transitions.

1. *InputCycleInitialize* — this transition initializes the state machine.
2. *StartReadInputChannel* — this transition starts the processing of input.
3. *ReadInputChannel* — this transition is a very complex composite transition. With each step, it reads one part of any of the three types of *InputStructures*.
4. *InputCycleFinish* — this transition finishes the *InputCycle* when all of the input has been read.
5. *InputCycleReset* — this transition resets the state machine.

6.6.1 Legal Input

The *InputCycle* is the only place in the specification of Soar that an error condition is checked. An example of an errorful input is an input structure that attempts to remove working memory elements that are not in the transitive closure of its input attribute.

The specification can check that its input sequence is correct before execution, but as the implementation does not have the full sequence of inputs available at start up time, it must check them one by one. For consistency we have specified that each cycle of input is checked for correctness as they are popped from the sequence of inputs. This subsection defines the legal input predicate to test each input

An identifier is an *InputLink* of an attribute, if it there is an augmentation of the top state in working memory that holds it as a value.

$$\begin{array}{l}
\mathit{InputLink\_} : \mathbb{P}\ \mathit{WME} \times \mathit{Identifier} \times \mathit{InputAttribute} \leftrightarrow \mathit{Identifier} \\
\hline
\forall\ \mathit{WM} : \mathbb{P}\ \mathit{WME};\ s : \mathit{Identifier};\ \mathit{InputAttribute} : \mathit{InputAttribute}; \\
\quad \mathit{id} : \mathit{Identifier};\ \mathit{link} : \mathit{Preference} \bullet \\
\quad \mathit{InputLink}((\mathit{WM}, s, \mathit{InputAttribute}), \mathit{id}) \Leftrightarrow \\
\qquad (\exists\ w : \mathit{WM} \bullet w.\mathit{id} = s \land w.\mathit{attribute} = \mathit{InputAttribute} \land \\
\qquad\quad w.\mathit{value} = \mathit{id} \land w.\mathit{context\_acceptable\_preference} = \mathit{No})
\end{array}$$

LegalInput checks the first *CycleOfInput* in the *inputs* sequence for several properties:

- that no new and modify are for the same object
- that no new and delete are for the same object
- that no delete and modify are for the same object
- that no two news are for the same object
- that no two modifies are for the same object

- that the adds of different modifies are disjoint, and that one modify is not adding what another is deleting
- that a *NewInputStructure* is for an *InputLink* that does not exist on the top state
- that a *DeleteInputStructure* is for an existing *InputLink* and
- that a *ModifyInputStructure* is for an existing *InputLink*, and that its adds and deletes modify the transitive closure of the input object.

LegalInput

IO

$$\begin{array}{l}
\forall\, channel : InputChannel \bullet \\
\quad \forall\, is1, is2 : inputs(1)(channel) \bullet \\
\quad\quad is1 \in \operatorname{ran} New \wedge is2 \in \operatorname{ran} New \Rightarrow \\
\quad\quad\quad ((New^{\sim}(is1)).new\_object = (New^{\sim}(is2)).new\_object \Rightarrow is1 = is2) \wedge \\
\quad\quad is1 \in \operatorname{ran} Modify \wedge is2 \in \operatorname{ran} Modify \Rightarrow \\
\quad\quad\quad ((Modify^{\sim}(is1)).modify\_object = \\
\quad\quad\quad\quad (Modify^{\sim}(is2)).modify\_object \Rightarrow is1 = is2 \wedge \\
\quad\quad\quad \operatorname{disjoint} \langle (Modify^{\sim}(is1)).adds, (Modify^{\sim}(is2)).adds \rangle \wedge \\
\quad\quad\quad \operatorname{disjoint} \langle (Modify^{\sim}(is1)).adds, (Modify^{\sim}(is2)).deletes \rangle) \wedge \\
\quad\quad is1 \in \operatorname{ran} New \wedge is2 \in \operatorname{ran} Modify \Rightarrow \\
\quad\quad\quad ((New^{\sim}(is1)).new\_object \neq (Modify^{\sim}(is2)).modify\_object) \wedge \\
\quad\quad is1 \in \operatorname{ran} New \wedge is2 \in \operatorname{ran} Delete \Rightarrow \\
\quad\quad\quad ((New^{\sim}(is1)).new\_object \neq (Delete^{\sim}(is2)).delete\_object) \wedge \\
\quad\quad is1 \in \operatorname{ran} Delete \wedge is2 \in \operatorname{ran} Modify \Rightarrow \\
\quad\quad\quad ((Delete^{\sim}(is2)).delete\_object \neq (Modify^{\sim}(is2)).modify\_object)
\end{array}$$

$$\begin{array}{l}
\forall\, channel : InputChannel \bullet \forall\, is : inputs(1)(channel) \bullet \\
\quad is \in \operatorname{ran} New \Rightarrow \\
\quad\quad (\neg\, InputLink((WM, TopState, InputMapping(channel)), \\
\quad\quad\quad\quad (New^{\sim}(is)).new\_object)) \wedge \\
\quad is \in \operatorname{ran} Delete \Rightarrow \\
\quad\quad InputLink((WM, TopState, InputMapping(channel)), \\
\quad\quad\quad\quad (Delete^{\sim}(is)).delete\_object) \wedge \\
\quad is \in \operatorname{ran} Modify \Rightarrow \\
\quad\quad (InputLink((WM, TopState, InputMapping(channel)), \\
\quad\quad\quad\quad (Modify^{\sim}(is)).modify\_object) \wedge \\
\quad\quad ((Modify^{\sim}(is)).deletes \\
\quad\quad\quad \subseteq OfId_{WME}((TCI_{WME}(WM))(\{(Modify^{\sim}(is)).modify\_object\}), WM) \wedge \\
\quad\quad ((Modify^{\sim}(is)).adds \\
\quad\quad\quad \subseteq OfId_{WME}((TCI_{WME}(WM \cup (Modify^{\sim}(is)).adds)) \\
\quad\quad\quad\quad (\{(Modify^{\sim}(is)).modify\_object\}), WM \cup (Modify^{\sim}(is)).adds))))
\end{array}$$

Although the specification defines the conditions of acceptable input, it does not specify how the implementation should handle unacceptable input.

6.6.2 Initializing and Starting the *InputCycle*

InputCycleInitialize moves the *InputCycle* state machine into its initial state.

$$
\begin{array}{l}
\textit{InputCycleInitialize} \\
\quad \Delta IO \\
\hline
\quad \textit{input\_cycle\_state} = \textit{InputCycleFinishedState} \\
\quad \textit{input\_cycle\_state}' = \textit{InputCycleInitialState} \\
\hline
\end{array}
$$

StartReadInput moves the machine into the *InputCycleReadState* to begin reading input. The whole *CycleOfInput* is checked for legality using the *LegalInput* predicate.

$$
\begin{array}{l}
\textit{StartReadInput} \\
\quad \Delta IO \\
\hline
\quad \textit{LegalInput} \\
\quad \textit{input\_cycle\_state} = \textit{InputCycleInitialState} \\
\quad \textit{input\_cycle\_state}' = \textit{InputCycleReadState} \\
\hline
\end{array}
$$

6.6.3 Reading NewInputStructures

When the first *CycleOfInput* holds a *NewInputStructure*, *ReadNewInputStructureObject* reads in its object. Chunking's *TraceWME* operation is called to give the working memory element a new instance number.

$$
\begin{array}{l}
\textit{ReadNewInputStructureObject} \\
\quad \Delta IO \\
\hline
\quad \textit{input\_cycle\_state} = \textit{InputCycleReadState} = \textit{input\_cycle\_state}' \\
\quad \exists\, ic : \textit{InputChannel} \mid ic \in \mathrm{dom}(\textit{inputs}(1)) \bullet \\
\quad\quad \exists\, ia : \textit{InputAttribute} \mid ia = \textit{InputMapping}(ic) \bullet \\
\quad\quad\quad \exists\, n : \textit{inputs}(1)(ic) \cap \mathrm{ran}\,\textit{New};\ w : \textit{WME} \mid \\
\quad\quad\quad\quad \textit{makeWME}(\textit{TopState}, ic, (\textit{New}^{\sim}(n)).\textit{new\_object}, \textit{No}) \notin \textit{WM} \wedge \\
\quad\quad\quad\quad w = \textit{makeWME}(\textit{TopState}, ic, (\textit{New}^{\sim}(n)).\textit{new\_object}, \textit{No})\} \bullet \\
\quad\quad\quad (\textit{WM}' = \textit{WM} \cup \{w\} \wedge \\
\quad\quad\quad \textit{TraceWME}) \\
\hline
\end{array}
$$

After the top state has been augmented with the object, *ReadNewInputStructureWME* reads in one of the new working memory elements, and destructively removes it from the *NewInputStructure*.

ReadNewInputStructureWME

ΔIO

$input\_cycle\_state = InputCycleReadState = input\_cycle\_state'$

$\exists ic : InputChannel \mid ic \in \mathrm{dom}(inputs(1)) \bullet$
$\quad \exists ia : InputAttribute \mid ia = InputMapping(ic) \bullet$
$\quad\quad \exists n : inputs(1)(ic) \cap \mathrm{ran}\ New \bullet$
$\quad\quad\quad \exists N : NewInputStructure \mid N = New^{\sim}(n) \land$
$\quad\quad\quad\quad makeWME(TopState, ia, N.new\_object, No) \in WM \bullet$
$\quad\quad\quad \exists w : N.wmes \mid w \notin WM \bullet$
$\quad\quad\quad\quad \exists N' : NewInputStructure \mid$
$\quad\quad\quad\quad\quad N'.new\_object = N.new\_object \land$
$\quad\quad\quad\quad\quad N'.wmes = N.wmes \setminus \{w\} \bullet$
$\quad\quad\quad\quad (WM' = WM \cup \{w\} \land$
$\quad\quad\quad\quad TraceWME \land$
$\quad\quad\quad\quad inputs'(1) = inputs(1) \oplus$
$\quad\quad\quad\quad\quad \{ic \mapsto (inputs(1)(ic) \setminus \{n\}) \cup \{New(N')\}\} \land$
$\quad\quad\quad\quad tail(inputs') = tail(inputs))$

When all of the working memory elements have been read in, *ReadNewInputStructureDone* pops the input from the first cycle of input.

ReadNewInputStructureDone

ΔIO

$input\_cycle\_state = InputCycleReadState = input\_cycle\_state'$

$\exists ic : InputChannel \mid ic \in \mathrm{dom}(inputs(1)) \bullet$
$\quad \exists ia : InputAttribute \mid ia = InputMapping(ic) \bullet$
$\quad\quad \exists n : inputs(1)(ic) \cap \mathrm{ran}\ New \bullet$
$\quad\quad\quad \exists N : NewInputStructure \mid N = New^{\sim}(n) \land$
$\quad\quad\quad\quad makeWME(TopState, ia, N.new\_object, No) \in WM \bullet$
$\quad\quad\quad ((N.wmes = \varnothing \land$
$\quad\quad\quad\quad inputs'(1) = inputs(1) \oplus \{ic \mapsto inputs(1)(ic) \setminus \{n\}\}) \land$
$\quad\quad\quad\quad tail(inputs') = tail(inputs))$

ReadNewInputStructure is the composite operation for reading an entire *NewInputStructure*.

$$\begin{array}{l}
ReadNewInputStructure \mathrel{\widehat{=}} ReadNewInputStructureObject \lor \\
\qquad ReadNewInputStructureWME \lor \\
\qquad ReadNewInputStructureDone
\end{array}$$

6.6.4 Reading ModifyInputStructures

When there is a *ModifyInputStructure* in the first *CycleOfInput*, *ReadModifyInputStructureAdd* reads in one of the elements that it should add, and deletes the element from the set of adds.

$$\begin{array}{l}
\text{— } ReadModifyInputStructureAdd \text{ —} \\
\Delta IO \\
\hline
input\_cycle\_state = InputCycleReadState = input\_cycle\_state' \\
\exists\, ic : InputCh\text{[illegible]} \mid ic \in \operatorname{dom}(inputs(1)) \bullet \\
\quad \exists\, ia : Input\text{[illegible]}tribute \mid ia = InputMapping(ic) \bullet \\
\qquad \exists\, m\text{[illegible]}puts(1)(ic) \cap \operatorname{ran} New \bullet \\
\qquad\quad \exists\, M : ModifyInputStructure \mid M = Modify^{\sim}(m) \land \\
\qquad\qquad makeWME(TopState, ia, M.modify\_object, No) \in WM \bullet \\
\qquad\quad \exists\, w : M.adds \mid w \notin WM \bullet \\
\qquad\qquad \exists\, M' : ModifyInputStructure \mid \\
\qquad\qquad\quad M'.modify\_object = M.modify\_object \land \\
\qquad\qquad\quad M'.adds = M'.adds \setminus \{w\} \land \\
\qquad\qquad\quad M'.deletes = M.deletes \bullet \\
\qquad\qquad (WM' = WM \cup \{w\} \land \\
\qquad\qquad TraceWME \land \\
\qquad\qquad inputs'(1) = inputs(1) \oplus \\
\qquad\qquad\quad \{ic \mapsto (inputs(1)(ic) \setminus \{m\}) \cup \{Modify(M')\}\} \land \\
\qquad\qquad tail(inputs') = tail(inputs)) \\
\hline
\end{array}$$

ReadModifyInputStructureDelete reads in a delete of a modify, and removes the element from the modify's delete set.

$$
\begin{array}{l}
\textit{ReadModifyInputStructureDelete} \\
\hline
\Delta IO \\
\hline
\mathit{input\_cycle\_state} = \mathit{InputCycleReadState} = \mathit{input\_cycle\_state}' \\[4pt]
\exists\, ic : \mathit{InputChannel} \mid ic \in \operatorname{dom}(\mathbf{inputs}(1)) \bullet \\
\quad \exists\, ia : \mathit{InputAttribute} \mid ia = \mathit{InputMapping}(ic) \bullet \\
\quad\quad \exists\, m : \mathbf{inputs}(1)(ic) \cap \operatorname{ran} \mathit{New} \bullet \\
\quad\quad\quad\quad \exists\, M : \mathit{ModifyInputStructure} \mid M = \mathit{Modify}^{\sim}(m) \wedge \\
\quad\quad\quad\quad \mathit{makeWME}(\mathit{TopState}, ia, M.\mathit{modify\_object}, \mathit{No}) \in \mathit{WM} \bullet \\
\quad\quad\quad \exists\, w : M.\mathbf{deletes} \mid w \in \mathit{WM} \bullet \\
\quad\quad\quad \exists\, M' : \mathit{ModifyInputStructure} \mid \\
\quad\quad\quad\quad M'.\mathit{modify\_object} = M.\mathit{modify\_object} \wedge \\
\quad\quad\quad\quad M'.\mathbf{deletes} = M'.\mathbf{deletes} \setminus \{w\} \wedge \\
\quad\quad\quad\quad M'.\mathbf{adds} = M.\mathbf{adds} \bullet \\
\quad\quad\quad (\mathit{WM}' = \mathit{WM} \setminus \{w\} \wedge \\
\quad\quad\quad\; \mathbf{inputs}'(1) = \mathbf{inputs}(1) \oplus \\
\quad\quad\quad\quad \{ic \mapsto \mathbf{inputs}(1)(ic) \setminus \{m\}\} \wedge \\
\quad\quad\quad\; \mathbf{tail}(\mathbf{inputs}') = \mathbf{tail}(\mathbf{inputs})) \\
\hline
\end{array}
$$

When all of the changes have been processed, *ReadModifyInputStructure-Done* pops the modify from the first cycle of inputs.

$$
\begin{array}{l}
\textit{ReadModifyInputStructureDone} \\
\hline
\Delta IO \\
\hline
\mathit{input\_cycle\_state} = \mathit{InputCycleReadState} = \mathit{input\_cycle\_state}' \\[4pt]
\exists\, ic : \mathit{InputChannel} \mid ic \in \operatorname{dom}(\mathbf{inputs}(1)) \bullet \\
\quad \exists\, ia : \mathit{InputAttribute} \mid ia = \mathit{InputMapping}(ic) \bullet \\
\quad\quad \exists\, m : \mathbf{inputs}(1)(ic) \cap \operatorname{ran} \mathit{New} \bullet \\
\quad\quad\quad \exists\, M : \mathit{ModifyInputStructure} \mid \\
\quad\quad\quad\quad\quad M = \mathit{Modify}^{\sim}(m) \wedge M.\mathbf{adds} = M.\mathbf{deletes} = \varnothing \bullet \\
\quad\quad\quad\quad (\mathbf{inputs}'(1) = \mathbf{inputs}(1) \oplus \{ic \mapsto \mathbf{inputs}(1)(ic) \setminus \{m\}\} \wedge \\
\quad\quad\quad\quad\; \mathbf{tail}(\mathbf{inputs}') = \mathbf{tail}(\mathbf{inputs})) \\
\hline
\end{array}
$$

ReadModifyInputStructure is the composite operation that joins all of the operations for reading a *ModifyInputStructure*.

$$
\begin{array}{l}
\mathit{ReadModifyInputStructure} \mathrel{\widehat{=}} \mathit{ReadModifyInputStructureAdd} \vee \\
\quad\quad\quad\quad \mathit{ReadModifyInputStructureDelete} \vee \\
\quad\quad\quad\quad \mathit{ReadModifyInputStructureDone}
\end{array}
$$

6.6.5 Reading DeleteInputStructures

ReadDeleteInputStructure pops a *DeleteInputStructure* from the first *CycleOfInput*, and removes the associated *InputLink*. The working memory elements in the transitive closure of the link are not explicitly deleted. The working memory elements that are no longer accessible from the context stack will eventually be removed by decide.

$$
\begin{array}{l}
\textit{ReadDeleteInputStructure} \\
\hline
\Delta IO \\
\hline
input\_cycle\_state = InputCycleReadState = input\_cycle\_state' \\[6pt]
\exists\, ic : InputChannel \mid ic \in \mathrm{dom}(inputs(1)) \bullet \\
\quad \exists\, ia : InputAttribute \mid ia = InputMapping(ic) \bullet \\
\quad\quad \exists\, d : inputs(1)(ic) \cap \mathrm{ran}\ Delete \bullet \\
\quad\quad\quad (WM' = WM \setminus \\
\quad\quad\quad\quad \{makeWME(TopState, ia, (Delete^{\sim}(d)).delete\_object, No)\} \wedge \\
\quad\quad\quad\; inputs'(1) = inputs(1) \oplus \\
\quad\quad\quad\quad \{ic \mapsto inputs(1)(ic) \setminus \{d\}\} \wedge \\
\quad\quad\quad\; tail(inputs') = tail(inputs)) \\
\hline
\end{array}
$$

6.6.6 Reading an Input Channel

CloseInputChannel will close any channel that has no remaining input structures.

$$
\begin{array}{l}
\textit{CloseInputChannel} \\
\hline
\Delta IO \\
\hline
input\_cycle\_state = InputCycleReadState = input\_cycle\_state' \\[6pt]
TopState \neq \text{`NIL'} \\[6pt]
\exists\, ic : InputChannel \mid ic \in \mathrm{dom}(inputs(1)) \bullet \\
\quad (inputs(1)(ic) = \emptyset \wedge \\
\quad\; inputs'(1) = \{ic\} \lhd inputs(1) \wedge \\
\quad\; tail(inputs') = tail(inputs)) \\
\hline
\end{array}
$$

ReadInputChannelAnyInputStructure will take one step of reading any input structure of the first cycle of input.

$$\begin{aligned}
&\mathit{ReadInputChannel} \mathrel{\widehat{=}} \mathit{ReadNewInputStructure} \lor \\
&\qquad \mathit{ReadDeleteInputStructure} \lor \\
&\qquad \mathit{ReadModifyInputStructure} \lor \\
&\qquad \mathit{CloseInputChannel}
\end{aligned}$$

6.6.7 Finishing, Stepping and Resetting the *InputCycle*

When there are no more input structures to read on any input channels, *InputCycleFinish* pops the inputs vector and moves the input cycle to the finished state.

```
── InputCycleFinish ──────────────────────────
 ΔIO
──────────────
 input_cycle_state = InputCycleFinishedState
 ¬ pre ReadInputChannel
 inputs' = tail(inputs)
 input_cycle_state' = InputCycleFinishedState
──────────────────────────────────────────────
```

InputCycleStep steps the input cycle using the three operations.

$$\begin{aligned}
&\mathit{InputCycleStep} \mathrel{\widehat{=}} \mathit{StartReadInput} \lor \mathit{ReadInputChannel} \lor \\
&\qquad \mathit{InputCycleFinish}
\end{aligned}$$

InputCycleReset is specified to allow the implementation to reset the operation of the input cycle state machine at any point of execution.

```
── InputCycleReset ───────────────────────────
 ΔIO
──────────────
 input_cycle_state' = InputCycleInitialState
──────────────────────────────────────────────
```

6.7 The Output Cycle

The *OutputCycle* state machine requires only five transitions.

1. *OutputChannelInitialize* — this operation initializes the *OutputCycle* machine.
2. *StartSendOutputChannel* — this operation moves the *OutputCycle* from the initial state into the state that sends out output.
3. *SendOutputChannel* — this composite operation ships an *OutputStructure* out an *OutputChannel*.
4. *OutputCycleFinish* — when there are no more outputs to send, this operation ends the execution of the state machine.
5. *OutputCycleReset* — this operation allows the implementation to reset the output cycle state machine when unusual conditions occur.

OutputCycleInitialize initializes the *OutputCycle* state machine by moving it from its finished, to its initial state.

OutputCycleInitialize

ΔIO

$output\_cycle\_state = OutputCycleFinishedState$

$output\_cycle\_state' = OutputCycleInitialState$

StartSendOutputChannel starts the shipping of output by moving the machine into the *OutputCycleSendState* state and pushing an empty *CycleOfOutputs* onto the end of the *outputs* sequence.

StartSendOutputChannel

ΔIO

$output\_cycle\_state = OutputCycleInitialState$

$output\_cycle\_state' = OutputCycleSendState$

$outputs' = outputs \frown \langle \varnothing \rangle$

If an *OutputLink* of the top level state does not have a corresponding *OutputStructure* in the last *CycleOfOutput*, then *SendOutputChannelOutputStructureObject* creates a new *OutputStructure* with the object of the link and adds it to the output.

$$
\begin{array}{l}
\textit{SendOutputChannelOutputStructureObject} \\
\hline
\Delta \mathit{IO} \\
\hline
\mathit{output\_cycle\_state} = \mathit{OutputCycleSendState} = \mathit{output\_cycle\_state}' \\[4pt]
\exists\, i : \mathit{OutputAttribute} \bullet \\
\quad \exists\, w : \mathit{WM} \mid w.\mathit{id} = \mathit{TopState} \land w.\mathit{attribute} = i \land \\
\qquad w.\mathit{context\_acceptable\_preference} = \mathit{No} \land \\
\qquad \mathit{OutputMapping}(i) \notin \mathrm{dom}(\mathit{last}(\mathit{outputs})) \bullet \\
\quad \exists\, \mathit{OS} : \mathit{OutputStructure} \mid \\
\qquad\quad \mathit{OS}.\mathit{object} = w.\mathit{value} \land \mathit{OS}.\mathit{wmes} = \varnothing \bullet \\
\qquad (\mathit{last}(\mathit{outputs}') = \mathit{last}(\mathit{outputs}) \oplus \\
\qquad\quad \{\mathit{OutputMapping}(i) \mapsto \mathit{OS}\} \land \\
\qquad \mathit{front}(\mathit{outputs}') = \mathit{front}(\mathit{outputs})) \\
\hline
\end{array}
$$

SendOutputChannelOutputStructureWME finds any working memory elements of the transitive closure of an *OutputLink* that have not been added to their *OutputStructure* and adds them to the structure.

$$
\begin{array}{l}
\textit{SendOutputChannelOutputStructureWME} \\
\hline
\Delta \mathit{IO} \\
\hline
\mathit{output\_cycle\_state} = \mathit{OutputCycleSendState} = \mathit{output\_cycle\_state}' \\[4pt]
\exists\, i : \mathit{OutputAttribute} \bullet \\
\quad \exists\, w : \mathit{WM} \mid w.\mathit{id} = \mathit{TopState} \land w.\mathit{attribute} = i \land \\
\qquad w.\mathit{context\_acceptable\_preference} = \mathit{No} \bullet \\
\qquad \exists\, \mathit{OS}, \mathit{OS}' : \mathit{OutputStructure} \mid \\
\qquad\qquad \mathit{OS}.\mathit{object} = w.\mathit{value} \land \mathit{OS}'.\mathit{object} = w.\mathit{value} \bullet \\
\qquad\quad \exists\, w_2 : \mathit{OfId}_{\mathit{WME}}(\mathit{TCI}_{\mathit{WME}}(\mathit{WM})(\{w.\mathit{value}\}), \mathit{WM}) \mid \\
\qquad\qquad w_2 \notin \mathit{OS}.\mathit{wmes} \land \mathit{OS}'.\mathit{wmes} = \mathit{OS}.\mathit{wmes} \cup \{w_2\} \bullet \\
\qquad\qquad (\mathit{last}(\mathit{outputs}') = \mathit{last}(\mathit{outputs}) \oplus \\
\qquad\qquad\quad \{\mathit{OutputMapping}(i) \mapsto \mathit{OS}'\} \land \\
\qquad\qquad \mathit{front}(\mathit{outputs}') = \mathit{front}(\mathit{outputs})) \\
\hline
\end{array}
$$

SendOutputChannel composes the operations to send out *OutputStructures*.

$$
\begin{array}{l}
\mathit{SendOutputChannel} \mathrel{\widehat=} \mathit{SendOutputChannelOutputStructureObject} \lor \\
\qquad\qquad \mathit{SendOutputChannelOutputStructureWME}
\end{array}
$$

When there is no more output to send, *OutputCycleFinish* finishes the output cycle.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{OutputCycleFinish}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{IO} \\
\hline
\mathit{output\_cycle\_state} = \mathit{OutputCycleSendState} \\
\neg\ \mathrm{pre}\ \mathit{SendOutputChannel} \\
\mathit{output\_cycle\_state}' = \mathit{OutputCycleFinishedState} \\
\hline
\end{array}$$

OutputCycleStep sequences all of the operations of the *OutputCycle*.

$$\begin{array}{l}
\mathit{OutputCycleStep} \mathrel{\widehat{=}} \\
\quad \mathit{StartSendOutputChannel} \lor \mathit{SendOutputChannel} \lor \\
\quad\quad \mathit{OutputCycleFinish}
\end{array}$$

OutputCycleReset allows the implementation to reset the *OutputCycle* state machine from any state under exceptional conditions.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{OutputCycleReset}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{IO} \\
\hline
\mathit{output\_cycle\_state}' = \mathit{OutputCycleInitialState} \\
\hline
\end{array}$$

Chapter 7

Decide

The decision procedure reads preference memory to decide what should be in working memory, including the context impasse stack and attribute impasses. Decide's specification is composed of a state schema and four state machines: impasser, preference semantics, working memory phase and quiescence phase. Soar's top level state machine steps decide through the working memory phase (WMPhase) or the quiescence phase (QPhase). The working memory phase and the quiescence phase step preference semantics to determine the values of slots and the impasser to add, modify and remove goals and impasses.

Z's bottom-up approach to specification requires that we first specify the state schema of a sequential system, and then specify its transitions. Consequently we present the state schemas for decide's four state machines first, then we present decide's total state schema, and then we present the state transitions for each of the component state machines.

This chapter is organized into ten sections.

- Section 7.1 Impasses — defines impasses, Soar's goal representation.
- Section 7.2 Impasser State - defines the state of the impasser.
- Section 7.3 Preference Semantics State — defines the state of preference semantics.
- Section 7.4 Working Memory Phase State — defines the state of working memory phase.
- Section 7.5 Quiescence Phase State — defines the state of quiescence phase.
- Section 7.6 Decide's Total State — defines the total state of decide.
- Section 7.7 Impasser — defines the state machine that implements impassing.

- Section 7.8 Preference Semantics — defines the state machine that implements preference semantics calculations.
- Section 7.9 Working Memory Phase — defines the state machine that implements the working memory phase.
- Section 7.10 Quiescence Phase — defines the state machine that implements quiescence phase.

7.1 Impasses

This section describes the basic concepts of an impasse, and provides some operations to help access and modify the working memory elements of impasses.

The context impasse stack is represented as elements in working memory, and so are its attribute impasses. Each element of a context impasse in the stack holds the context's identifier in its id component, and its attribute is an element of *ImpasseAttribute*, defined in Section 5.5.9.

$$ImpassesWMEs : Symbol \times \mathbb{P}\ Identifier \times \mathbb{P}\ WME \to \mathbb{P}\ WME$$

$$\begin{array}{l} \forall\, g\_or\_i : Symbol;\ GM : \mathbb{P}\ Identifier;\ WM : \mathbb{P}\ WME \bullet \\ \quad ImpassesWMEs(g\_or\_i, GM, WM) = \\ \quad\quad \{w : WM \mid g\_or\_i \in GM \land w.id = g\_or\_i \land \\ \quad\quad\quad w.attribute \in ImpasseAttribute \land \\ \quad\quad\quad w.context\_acceptable\_preference = No\} \end{array}$$

There are eight attributes that are used to create working memory elements defining a context impasse.

1. `PROBLEM-SPACE' — the optional problem space of the subgoal.
2. `STATE' — the optional state of the subgoal.
3. `OPERATOR' — the optional operator of the subgoal.
4. `OBJECT' — the parent context impasse for a context impasse, or the identifier of the impassed slot for an attribute impasse.
5. `ATTRIBUTE' — the attribute of the impassed slot.
6. `IMPASSE' — the type of impasse (see 7.2).
7. `ITEM' — the candidates that were available for decision for `TIE', `CONFLICT' or `CONSTRAINT-FAILURE' impasse.
8. `CHOICES' — if there are no `ITEM' augmentations, an impasse has a `CHOICES' augmentation of value `NONE', otherwise its value is `MULTIPLE'.

Decide uses only the 'OBJECT', 'ATTRIBUTE', 'IMPASSE', 'ITEM' and 'CHOICES' attributes for attribute impasses. The values from the ('OBJECT','ATTRIBUTE') working memory element pair form the slot for which decide had incomplete or inconsistent information. The 'IMPASSE' element holds as a value the type of impasse. The 'CHOICES' describes if there were 'MULTIPLE' or 'NONE' value choices available to the decision procedure for the impassed slot. Each choice is represented in working memory using an augmentation of attribute 'ITEM'.

The recursive function *GoalDescendents* follows the chain of 'OBJECT' attributes backwards through working memory to construct the set of identifiers of the descendant goals of a given goal.

$$GoalDescendants : Identifier \times \mathbb{P}\ Identifier \times \mathbb{P}\ WME \rightarrow \mathbb{P}\ Identifier$$

$$\begin{aligned}
&\forall g : Identifier;\ GM : \mathbb{P}\ Identifier;\ WM : \mathbb{P}\ WME \bullet \\
&\quad (\exists_1\ c : GM \bullet makeWME(c, \text{'OBJECT'}, g, No) \in WM) \Rightarrow \\
&\quad\quad (\exists_1\ c : GM \mid makeWME(c, \text{'OBJECT'}, g, No) \in WM \bullet \\
&\quad\quad\quad GoalDescendants(g, GM, WM) = \\
&\quad\quad\quad\quad \{c\} \cup GoalDescendants(c, GM, WM)) \wedge \\
&\quad \neg (\exists_1\ c : GM \bullet makeWME(c, \text{'OBJECT'}, g, No) \in WM) \Rightarrow \\
&\quad\quad GoalDescendants(g, GM, WM) = \varnothing
\end{aligned}$$

7.2 Impasser State

The impasser creates, modifies and deletes both context and attribute impasses. Its state machine requires only three states (Figure 7.1):

1. an initial state which is used to optionally remove old impasses
2. a state from which new impasses are created or old impasses have their items changed and
3. a final state.

$$\begin{aligned}
ImpasserState ::= &\ ImpasserInitialState \\
\mid &\ ImpasserCreateOrChangeState \\
\mid &\ ImpasserFinishedState
\end{aligned}$$

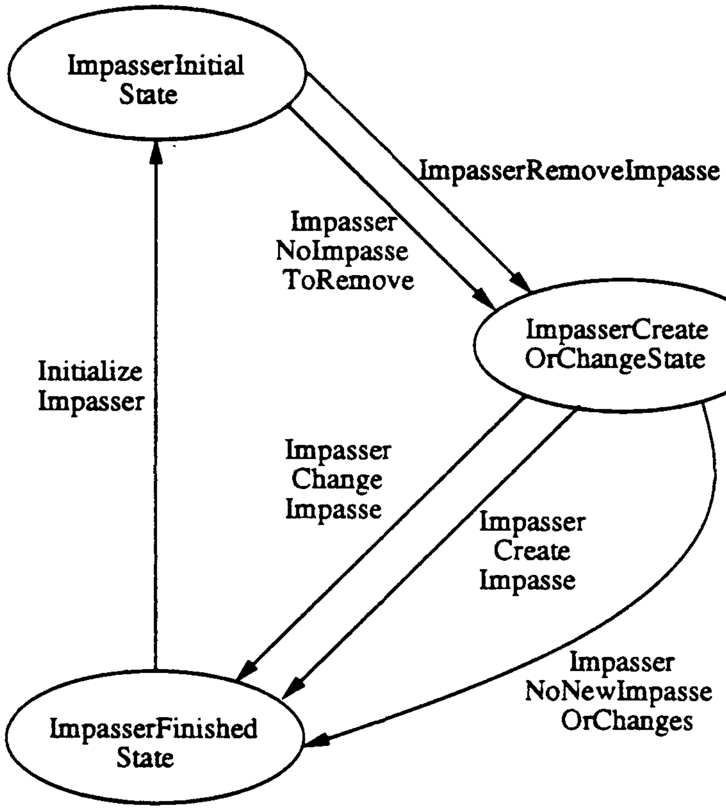

Figure 7.1: Impasser State Machine

The impasser is started with a slot that may need an impasse created, changed or removed, and the type of the impasse that is required.

$$
\begin{array}{|l}
ImpasseType : \mathbb{P}\ Constant \\
\hline
ImpasseType = \\
\quad \{\text{`CONSTRAINT-FAILURE'}, \text{`CONFLICT'}, \text{`TIE'}, \\
\quad \text{`NO-CHANGE'}, \text{`NONE'}\}
\end{array}
$$

There are four types of impasses and one symbol used to specify that the slot does not require an impasse.

1. `CONSTRAINT-FAILURE' - created when decide has more than one required candidate for a slot, or has a required candidate that is also prohibited.
2. `CONFLICT' — created when decide has inconsistent better, worse, best or worst preferences for a slot.
3. `TIE' — created when decide has more than one acceptable, non-parallel, non-indifferent candidate for a slot.
4. `NO-CHANGE' — created when quiescence is reached and the decision procedure has no change that it can make to the context stack.
5. `NONE' — signifies that preference semantics has determined that it has no candidate, or a unique candidate for a slot.

The flag, *context_changed*, of type *ContextChanged* is used to record if the context has been changed during quiescence phase or the impasser state machine. This allows the quiescence phase to know when the context has been changed so that it can stop looping over the slots of the context.

$ContextChanged ::= Changed \mid Unchanged$

The *ImpasserStateSchema* groups the slot to be impassed, the type of the impasse, the items under consideration from preference semantics and the state counter for the impasse state machine

$$
\begin{array}{|l}
\text{\_\_ } ImpasserStateSchema \text{ \_\_\_\_\_\_\_\_\_\_} \\
impasser\_state : ImpasserState \\
slot : Slot \\
impasse : ImpasseType \\
items : \mathbb{P}\ Symbol \\
context\_changed : ContextChanged \\
\hline
\end{array}
$$

7.3 Preference Semantics State

Preference semantics reads the contents of preference memory to decide new values for working memory slots. The preference semantics state machine requires eleven states (Figure 7.3): nine to perform the filter-like operations to decide slots, an initial and a final state.

1. PSInitialState — preference semantics starts out in this state.
2. PSRequireState — this state checks for required candidates in preference memory and also constraint failure impasses involving require preferences.
3. PSAcceptableState — this state collects the candidates that have acceptable preferences in memory.
4. PSProhibitState — this state filters out the candidates that have been prohibited.
5. PSRejectState — this state filters out the candidates that have rejects in PM.
6. PSBetterWorseState — this state checks for better worse conflicts, and otherwise filters out candidates based on better/worse preferences.
7. PSBestState — this state filters out candidates using the best preferences.
8. PSWorstState — this state filters out candidates using the worst preferences.
9. PSIndifferentState — If all of the remaining candidates are mutually indifferent, this state returns them as indifferent. If they are not, and this is a context impasse slot, a tie impasse is returned. Non-context slots's items are tested to see if they are mutually parallel.
10. PSParallelState — If all of the remaining candidates are mutually parallel, they're returned as mutually parallel. Otherwise, a tie impasse is returned.
11. PSFinishedState — preference semantics enters this state when it has reached a conclusion for its slot.

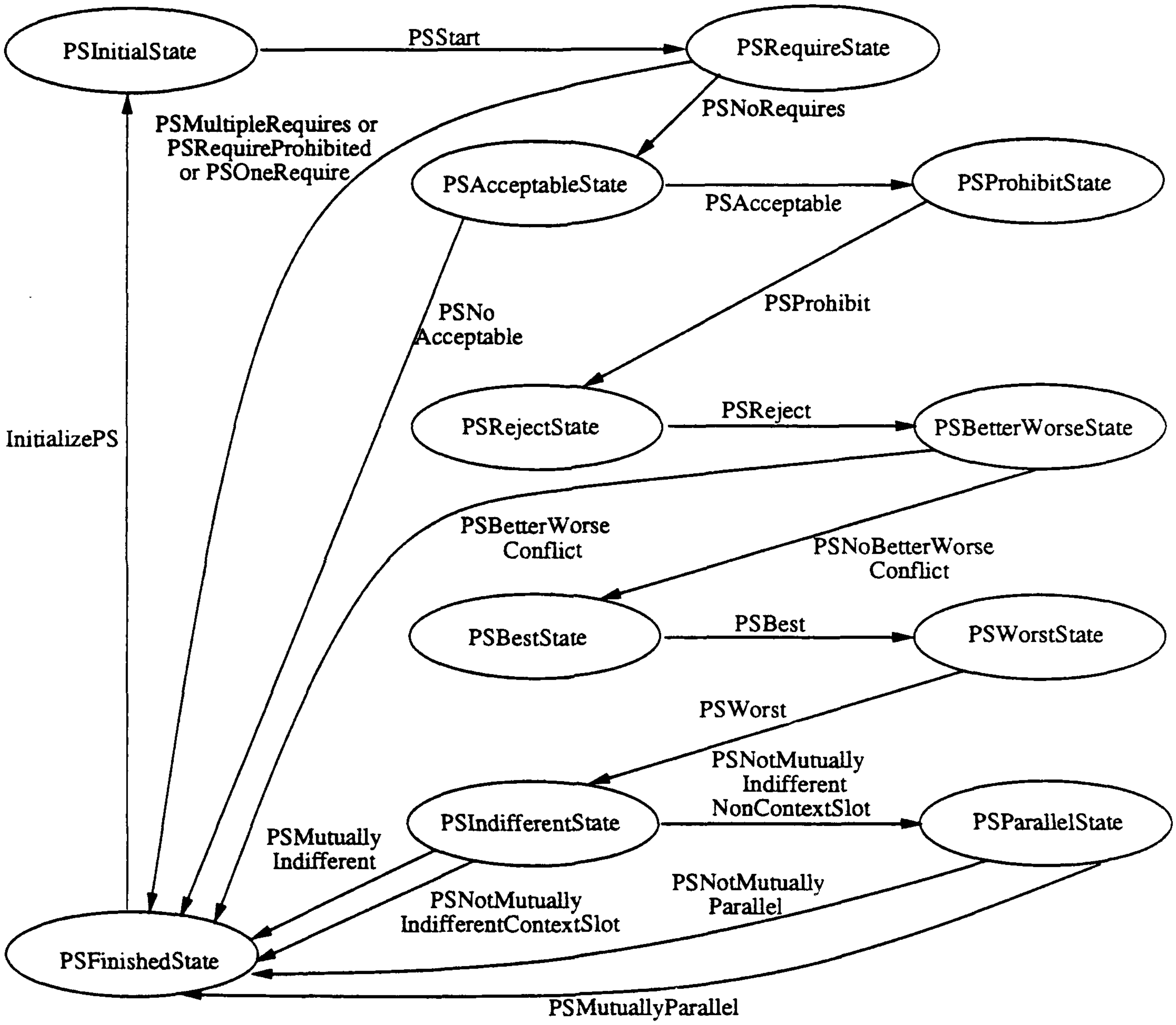

Figure 7.2: Preference Semantics State Machine

$$\begin{array}{rl} \mathit{PreferenceSemanticsState} ::= & \mathit{PSInitialState} \\ \mid & \mathit{PSRequireState} \\ \mid & \mathit{PSAcceptableState} \\ \mid & \mathit{PSProhibitState} \\ \mid & \mathit{PSRejectState} \\ \mid & \mathit{PSBetterWorseState} \\ \mid & \mathit{PSBestState} \\ \mid & \mathit{PSWorstState} \\ \mid & \mathit{PSIndifferentState} \\ \mid & \mathit{PSParallelState} \\ \mid & \mathit{PSFinishedState} \end{array}$$

Preference semantics requires an input slot, and produces an impasse, a possibly empty set of conflicting, indifferent or parallel items and a flag, number_of_winners of type *NumberOfWinners*. The number_of_winners flag is set whenever there is a winning value, e.g., there is not an impasse. If a single winner is chosen or there are a set of mutually indifferent winners. number_of_winners is *One*. If all of the winners are mutually parallel, then number_of_winners is *All*. This allows WMPhase to distinguish a set of mutually parallel candidates that should all be installed in working memory from a set of mutually indifferent candidates that should have only one value installed or maintained.

$$\mathit{NumberOfWinners} ::= \mathit{All} \mid \mathit{One}$$

The *PreferenceSemanticsStateSchema* schema groups preference semantics's inputs, outputs and state counter.

```
__ PreferenceSemanticsStateSchema ______________________
| ps_state : PreferenceSemanticsState
| slot : Slot
| items : P Symbol
| impasse : ImpasseType
| number_of_winners : NumberOfWinners
|________________________________________________________
```

7.4 Working Memory Phase State

The WMPhase state machine is implemented in six states (Figure 7.3).

1. *WMPhaseInitialState* — the state machine starts out in this state.

2. *WMPhasePickSlotState* — this state picks any non-context slot with changed preferences for decision.

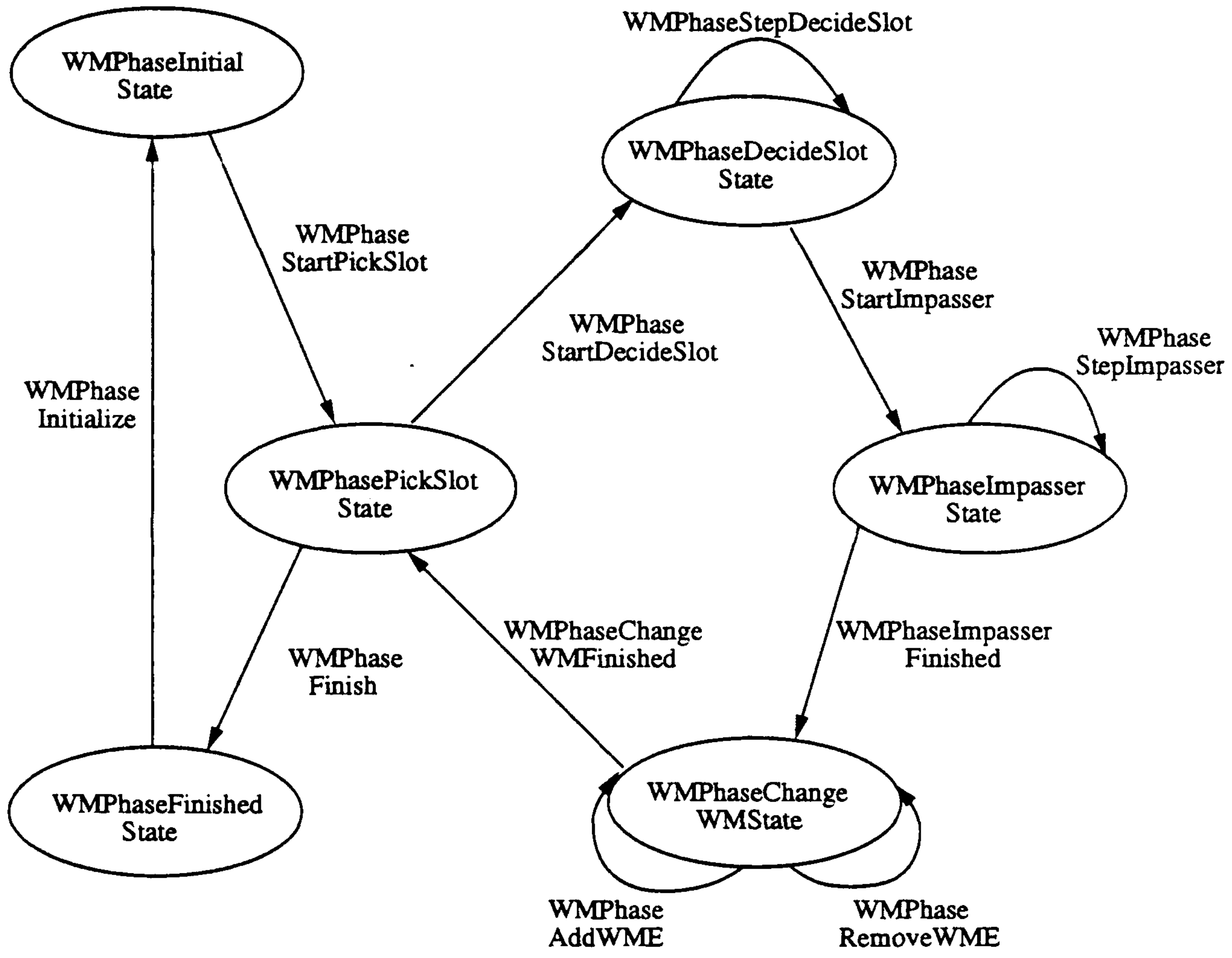

Figure 7.3: Working Memory Phase State Machine

3. *WMPhaseDecideSlotState* — this state steps the preference semantics state machine until it has arrived at a decision.

4. *WMPhaseImpasseState* — if the decision is an impasse, this state steps the impasse machine until is has added, changed or removed the impasse.

5. *WMPhaseChangeWMState* — this state adds or removes the WMEs for the slot.

6. *WMPhaseFinishedState* — the state machine finishes in this state.

$$
\begin{aligned}
WMPhaseState ::= {} & WMPhaseInitialState \\
| {} & WMPhasePickSlotState \\
| {} & WMPhaseDecideSlotState \\
| {} & WMPhaseImpasserState \\
| {} & WMPhaseChangeWMState \\
| {} & WMPhaseFinishedState
\end{aligned}
$$

The preference phase records the set of slots that have preferences added or retracted in the *ChangedSlots* component of the *WMPhaseStateSchema*. The WMPhase destructively iterates the component *slot* over the changed slots, runs the decision procedure on the *slot* and receives back information in the *items*, *impasse* and the *number_of_winners* components.

```
— WMPhaseStateSchema ————————————————————
| wmphase_state : WMPhaseState
| changed_slots : ℙ Slot
| slot : Slot
| items : ℙ Symbol
| impasse : ImpasseType
| number_of_winners : NumberOfWinners
————————————————————————————————————————
```

7.5 Quiescence Phase State

Quiescence Phase's state machine requires six states (Figure 7.4).

1. QPhaseInitialState — the state machine starts from this state.

2. QPhasePickSlotState — chooses the context impasses and which one of its slots to decide next.

3. QPhaseDecideSlotState — this state steps the preference semantics state machine given this slot as an argument.

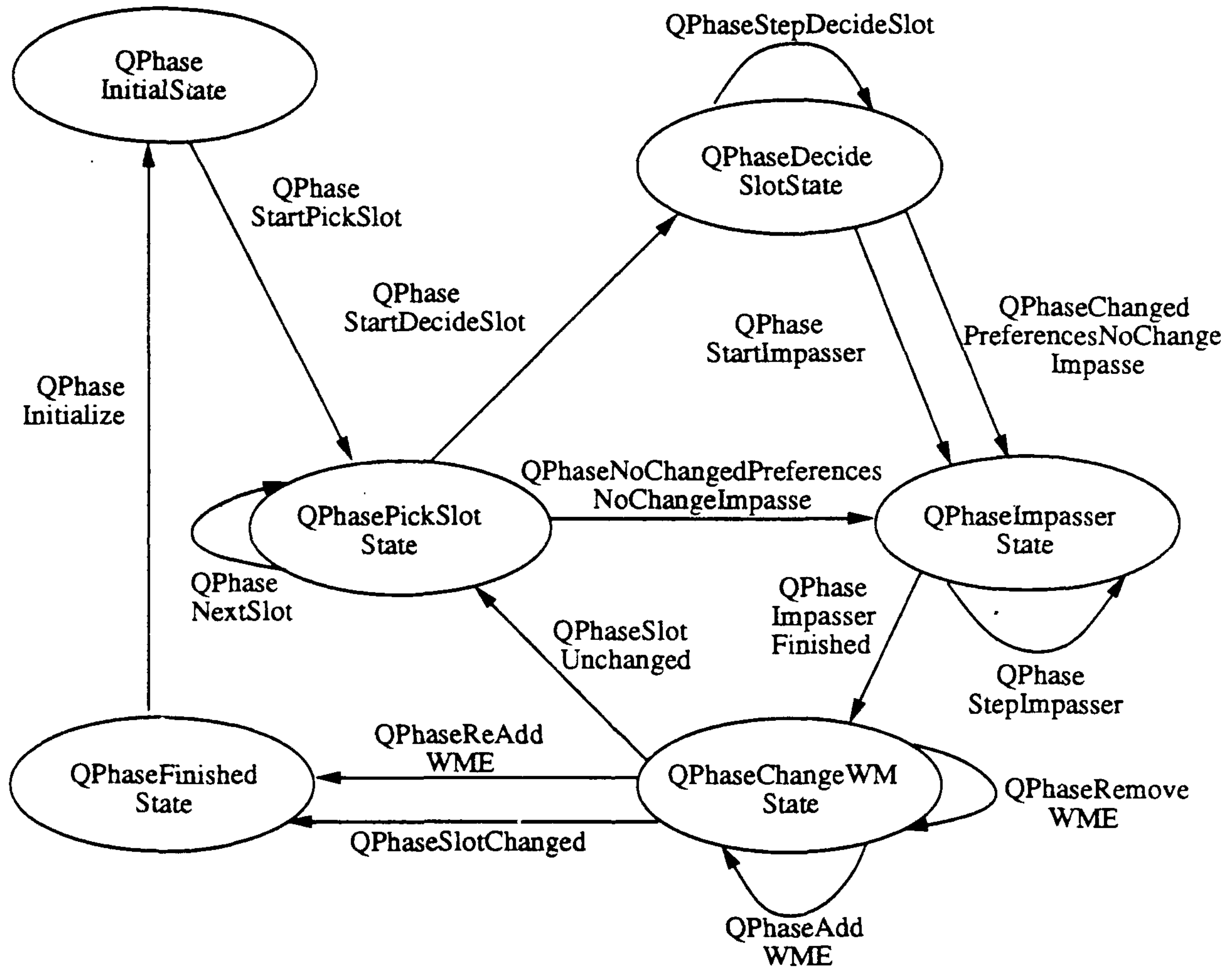

Figure 7.4: Quiescence Phase State Machine

4. QPhaseImpasserState — this state steps the impasser given the result of preference semantics.

5. QPhaseChangeWMState — this state changes the values in working memory for the slot.

6. QPhaseFinishedState — the final state of the machine.

$$\begin{aligned} QPhaseState ::= {} & QPhaseInitialState \\ \mid {} & QPhasePickSlotState \\ \mid {} & QPhaseDecideSlotState \\ \mid {} & QPhaseImpasserState \\ \mid {} & QPhaseChangeWMState \\ \mid {} & QPhaseFinishedState \end{aligned}$$

The set *ContextSlotAttribute* holds the attributes for context slots.

$$\begin{array}{|l} ContextSlotAttribute : \mathbb{P}\ ImpasseAttribute \\ \hline ContextSlotAttribute = \\ \quad \{\text{`PROBLEM-SPACE'}, \text{`STATE'}, \text{`OPERATOR'}\} \end{array}$$

The QPhase state machine uses six pieces of state:

1. a state counter, *qphase_state*

2. the slot of the context under consideration

3. a flag that marks if the context has been changed

4. the set of slots that preference phase has changed preferences for

5. the set of items that the preference semantics returned and

6. the number of winners amongst those items.

$$\begin{array}{|l} \hline \textit{QPhaseStateSchema} \\ \hline qphase\_state : QPhaseState \\ slot : Slot \\ context\_changed : ContextChanged \\ changed\_slots : \mathbb{P}\ Slot \\ items : \mathbb{P}\ Symbol \\ number\_of\_winners : NumberOfWinners \\ \hline \end{array}$$

7.6 Decide's Total State

The decide state schema includes the state schemas for its state machines, defines the impasse memory, the goal memory and two pointers into it, and shares the temporary memory, its component memories and the identifier table with the other major modules of Soar.

The goal memory, GM, and impasse memory, IM, are defined as sets of identifiers. All information about the type of context impasse or attribute impasse is represented by working memory elements. The goal memory is also constrained to be contained within the transitive closure of the bottom goal. Impasse memory is constrained somewhat differently: all of the objects of the attribute impasses must be connected to the bottom goal.

Temporary memory, TM, is defined and constrained by three recursive axioms.

1. Temporary memory is the union of the temporary memory elements created from all of working memory and production memory, using the free type constructors for temporary memory elements.

2. Temporary memory is defined to be transitively closed from the bottom goal and the impasses. This means that all elements are on a path that starts with the identifier of the bottom goal or an impasse identifier.

3. Working memory, which may be viewed as a part of temporary memory, contains a working memory element of *context_acceptable_preference Yes* if and only if there is a corresponding acceptable preference for that context slot. The mapping *ContextImpassePreferences* is used to construct the set of working memory elements that correspond to the context acceptable preferences.

$$\mathit{ContextImpassePreferences} : \mathbb{P}\ \mathit{Identifier} \times \mathbb{P}\ \mathit{Preference} \rightarrow \mathbb{P}\ \mathit{WME}$$

$$\begin{array}{l}
\forall\ GM : \mathbb{P}\ \mathit{Identifier};\ PM : \mathbb{P}\ \mathit{Preference} \bullet \\
\quad \mathit{ContextImpassePreferences}(GM, PM) = \\
\quad\quad \{p : \mathit{UnaryP}^{\sim}(\!|PM \cap \mathrm{ran}\ \mathit{UnaryP}|\!) \mid \\
\quad\quad\quad p.\mathit{id} \in GM \land (p.\mathit{preference} = `+' \lor p.\mathit{preference} = `!') \land \\
\quad\quad\quad p.\mathit{attribute} \in \\
\quad\quad\quad\quad \{`\mathrm{PROBLEM\text{-}SPACE}', `\mathrm{STATE}', `\mathrm{OPERATOR}'\} \\
\quad\quad\quad\quad \bullet\ \mathit{makeWME}(p.\mathit{id}, p.\mathit{attribute}, p.\mathit{value}, \mathit{Yes})\}
\end{array}$$

Decide

$IM : \mathbb{P}\ Identifier$
$GM : \mathbb{P}\ Identifier$
$bottom\_goal, top\_goal : Identifier$
$IdentifierTable : \mathbb{P}\ Identifier$
$TM : \mathbb{P}\ TME$
$WM : \mathbb{P}\ WME$
$PM : \mathbb{P}\ Preference$
$PreferenceSemanticsStateSchema$
$ImpasserStateSchema$
$WMPhaseStateSchema$
$QPhaseStateSchema$

$TM = PreferenceTME(\!|PM|\!) \cup WMETME(\!|WM|\!)$

$TM = OfId_{TME}((TCI_{TME}(TM))(\{bottom\_goal\} \cup IM), TM)$

$ContextImpassePreferences(GM, PM) =$
$\quad \{w : WM \mid w.context\_acceptable\_preference = Yes\}$

$IM = \{i : IM \mid$
$\quad \exists\, o : Symbol \bullet$
$\quad\quad WMETME(makeWME(i, \text{'OBJECT'}, o, No)) \in$
$\quad\quad\quad OfId_{TME}((TCI_{TME}(TM))(\{bottom\_goal\} \cup IM), TM)\}$

$GM = ((TCI_{TME}(TM))(\{bottom\_goal\})) \cap GM$

Decide starts off with its four state machines in their finished state, and empty impasse and goal memories.

InitDecide

$Decide$

$ps\_state = PSFinishedState$

$impasser\_state = ImpasserFinishedState$

$wmphase\_state = WMPhaseFinishedState$

$qphase\_state = QPhaseFinishedState$

$IM = \emptyset$

$GM = \{\}$

The top level state machine initializes decide with the *DecideCreateFirstGoal* operation. It sets goal memory to a singleton, adds the object working memory element for this goal and its type working memory element, and sets the top and bottom goal pointers to the single goal.

$$
\begin{array}{|l}
\hline
\textit{DecideCreateFirstGoal} \\
\Delta \textit{Decide} \\
\hline
\exists g : \textit{Identifier} \setminus \textit{IdentifierTable} \bullet \quad (GM = \{g\} \wedge \\
\quad \textit{top\_goal} = g \wedge \\
\quad \textit{bottom\_goal} = g \wedge \\
\quad WM' = \{\textit{makeWME}(g, \text{`OBJECT'}, \text{`NIL'}, \textit{No}), \\
\qquad \textit{makeWME}(g, \text{`TYPE'}, \text{`GOAL'}, \textit{No})\}) \\
\textit{ps\_state} = \textit{PSFinishedState} \\
\textit{impasser\_state} = \textit{ImpasserFinishedState} \\
\textit{wmphase\_state} = \textit{WMPhaseFinishedState} \\
\textit{qphase\_state} = \textit{QPhaseFinishedState} \\
IM = \varnothing \\
\hline
\end{array}
$$

7.7 Impasser Transitions

The impasser has six state transitions between its three states (Figure 7.1).

1. ImpasserInitialize — used to initialize the state machine.
2. ImpasserNoImpasseToRemove — If there is no need to remove an impasse, this transits the machine from the initial state to the CreateOrChange state.
3. ImpasserRemoveImpasse — If there is an old impasse in memory that is for the wrong type of impasse or the wrong slot, this transition removes it.
4. ImpasserCreateImpasse — If preference semantics has determined a need for an impasse that is not yet in memory, this creates it.
5. ImpasserChangeImpasse — If preference semantics has found a changed set of items for an existing `TIE', `CONFLICT' or `CONSTRAINT-FAILURE' impasse, this transition changes the items set.
6. ImpasserNoNewImpasseOrChanges — If there is not a new impasse and no changes to an item set, this finishes the state machine's execution.

ImpasserInitialize readies the impasser to run by moving it from the final to the initial state.

$$
\begin{array}{|l}
\textit{ImpasserInitialize} \\
\Delta Decide \\
\hline
context\_changed' = Unchanged \\
impasser\_state' = ImpasserInitialState
\end{array}
$$

The impasser first checks for an old impasse existing in memory that must be removed. An impasse must be removed if preference semantics has decided that the slot needs a different type of impasse, a higher slot in the context has an impasse, or the slot should have no impasse at all. *ImpasserRemoveImpasse* is designed to remove both context and non-context impasses, so it searches for an impasse identifier in IM or GM. If the obsolete impasse is a non-context impasse, it deletes it from IM. If the obsolete impasse is a context impasse, it deletes it and its descendants from GM, and updates the pointer to the bottom of the context stack.

$$
\begin{array}{|l}
\textit{ImpasserRemoveImpasse} \\
\Delta Decide \\
\hline
impasser\_state = ImpasserInitialState \\
\exists\, i : IM \cup GM \mid \\
\quad makeWME(i, \text{`OBJECT'}, first(slot), No) \in WM \bullet \\
\quad \neg\, ((makeWME(i, \text{`ATTRIBUTE'}, second(slot), No) \in WM) \wedge \\
\quad\;\; (makeWME(i, \text{`IMPASSE'}, impasse, No) \in WM)) \\
\quad \Rightarrow \\
\;\; ((i \in IM \Rightarrow \\
\quad IM' = IM \setminus \{i\} \wedge \\
\quad\quad WM' = WM \setminus ImpassesWMEs(i, IM, WM)) \wedge \\
\;\;\; (i \in GM \Rightarrow \\
\quad (context\_changed' = Changed \wedge \\
\quad (GM' = GM \setminus (\{i\} \cup GoalDescendants(i, GM, WM))) \wedge \\
\quad (bottom\_goal' = first(slot))) \wedge \\
\;\; (WM' = WM \setminus ImpassesWMEs(i, GM, WM) \\
\quad \setminus (\bigcup\{g : GoalDescendants(i, GM, WM) \bullet \\
\quad\quad\quad ImpassesWMEs(g, GM, WM)\})))) \\
impasser\_state' = ImpasserCreateOrChangeState
\end{array}
$$

If the impasse in memory is the desired type of impasse or there is no impasse in memory then *ImpasserNoImpasseToRemove* moves the impasser into the create or change state.

$$
\begin{array}{l}
\textbf{ImpasserNoImpasseToRemove} \\
\hline
\Delta Decide \\
\hline
impasser\_state = ImpasserInitialState \\
\exists i : IM \cup GM \bullet \\
\quad (makeWME(i, \text{`OBJECT'}, first(slot), No) \in WM \land \\
\quad makeWME(i, \text{`ATTRIBUTE'}, second(slot), No) \in WM \land \\
\quad makeWME(i, \text{`IMPASSE'}, impasse, No) \in WM) \lor \\
\neg (\exists i : IM \cup GM \bullet \\
\quad (makeWME(i, \text{`OBJECT'}, first(slot), No) \in WM \land \\
\quad makeWME(i, \text{`ATTRIBUTE'}, second(slot), No) \in WM)) \\
impasser\_state' = ImpasserCreateOrChangeState \\
\hline
\end{array}
$$

If the desired impasse is not in memory, *ImpasserCreateImpasse* selects an identifier that is not in the *IdentifierTable*, and consequently not in use by the system, and adds the identifier to IM or GM and the basic three elements required for the impasse. If the impasse is a 'CONSTRAINT-FAILURE' or a 'NO-CHANGE' then an element of attribute 'CHOICES' value 'NONE' is also added. If the impasse is a 'CONFLICT' or a 'TIE' then an element of attribute 'CHOICES' value 'MULTIPLE' is added instead. An element of attribute 'ITEM' is added for each candidate returned by preference semantics in the items set. Chunking's *TraceItems* operation is referenced to give the new item elements numbers and setup their working memory traces to point to the preferences that caused them to be considered for the slot.

```
ImpasserCreateImpasse
  $\Delta Decide$
  ----------------
  $impasser\_state = ImpasserCreateOrChangeState$

  $\neg\,(\exists\, i : IM \cup GM \bullet$
    $makeWME(i, \text{'OBJECT'}, first(slot), No) \in WM \wedge$
    $makeWME(i, \text{'ATTRIBUTE'}, second(slot), No) \in WM)$

  $\exists\, i : Identifier \mid i \notin IdentifierTable \bullet$
    $(first(slot) \in GM \wedge$
     $second(slot) \in ContextSlotAttribute \cup \{\text{'GOAL'}\}) \Rightarrow$
      $(GM' = GM \cup \{i\} \wedge$
      $context\_changed' = Changed \wedge$
      $bottom\_goal' = i) \wedge$
    $\neg\,(first(slot) \in GM \wedge$
     $second(slot) \in ContextSlotAttribute \cup \{\text{'GOAL'}\}) \Rightarrow$
      $IM' = IM \cup \{i\} \wedge$
    $\exists\, W : \mathbb{P}\, WME \mid W = \{v : items \bullet makeWME(i, \text{'ITEM'}, v, No)\}$
      $WM' = WM \cup$
        $\{makeWME(i, \text{'OBJECT'}, first(slot), No),$
          $makeWME(i, \text{'ATTRIBUTE'}, second(slot), No),$
          $makeWME(i, \text{'IMPASSE'}, impasse, No)\} \cup$
        $\{w : WME \mid w = makeWME(i, \text{'CHOICES'}, \text{'NONE'}, No) \wedge$
            $impasse \in \{\text{'CONSTRAINT-FAILURE'}, \text{'NO-CHANGE'}\}\} \cup$
        $\{w : WME \mid w = makeWME(i, \text{'CHOICES'}, \text{'MULTIPLE'}, No) \wedge$
            $impasse \in \{\text{'CONFLICT'}, \text{'TIE'}\}\} \cup W \wedge$
        $TraceItems$

  $impasser\_state' = ImpasserFinishedState$
```

If the correct type of impasse for the slot exists in memory, but its set of items is now different, *ChangeImpasse* removes the elements for the items that are no longer involved in the impasse, and adds in elements for the new items. This does not change the main structure of the context stack, only its items, so the context_changed flag is not set and sub-slots and sub-goals are not flushed. Having updated working memory to match preference semantic's required impasse, the *ChangeImpasse* steps to the finished impasser state.

ImpasserChangeImpasse

$$
\begin{array}{l}
\Delta Decide \\
\hline
impasser\_state = ImpasserCreateOrChangeState \\
\exists\, i : IM \cup GM \mid \\
\quad makeWME(i, \text{`OBJECT'}, first(slot), No) \in WM \wedge \\
\quad makeWME(i, \text{`ATTRIBUTE'}, second(slot), No) \in WM \wedge \\
\quad makeWME(i, \text{`IMPASSE'}, impasse, No) \in WM \wedge \\
\quad items \neq \{v : Symbol \mid makeWME(i, \text{`ITEM'}, v, No) \in WM\} \bullet \\
\quad \exists\, W : \mathbb{P}\, WME \mid \\
\qquad W = \{w : WM \mid \\
\qquad\quad \exists\, v : Symbol \setminus items \bullet w = makeWME(i, \text{`ITEM'}, v, No)\} \\
\qquad\quad \cup \{v : items \mid makeWME(i, \text{`ITEM'}, v, No) \notin WM \bullet \\
\qquad\qquad makeWME(i, \text{`ITEM'}, v, No)\} \bullet \\
\qquad WM' = WM \setminus W \ \wedge \qquad TraceItems \\
impasser\_state' = ImpasserFinishedState
\end{array}
$$

If there is no need for a new impasse, or the impasse in memory is exactly right, then *ImpasserNoNewImpasseOrChanges* completes the impasser's run.

ImpasserNoNewImpasseOrChanges

$$
\begin{array}{l}
\Delta Decide \\
\hline
impasser\_state = ImpasserCreateOrChangeState \\
impasse = \text{`NONE'} \vee \\
(\exists\, i : IM \cup GM \bullet \\
\quad makeWME(i, \text{`OBJECT'}, first(slot), No) \in WM \wedge \\
\quad makeWME(i, \text{`ATTRIBUTE'}, second(slot), No) \in WM \wedge \\
\quad makeWME(i, \text{`IMPASSE'}, impasse, No) \in WM \wedge \\
\quad items = \{v : Symbol \mid makeWME(i, \text{`ITEM'}, v, No) \in WM\}) \\
impasser\_state' = ImpasserFinishedState
\end{array}
$$

After initialization *ImpasserStep* will drive the impasser state machine through all of its states to completion.

$$\begin{aligned} ImpasserStep \mathrel{\widehat=}\ & ImpasserNoImpasseToRemove \lor ImpasserRemoveImpasse \lor \\ & ImpasserCreateImpasse \lor ImpasserChangeImpasse \lor \\ & ImpasserNoNewImpasseOrChanges \end{aligned}$$

7.8 Preference Semantics Transitions

Preference Semantics is defined in seventeen transitions (Figure 7.3): one to initialize, fifteen for processing the filter-like actions and one that is a concession to the obvious implementation to allow a fast exit from the state machine in a common case.

The *PSInitialize* initializes the state counter, empties the set of candidate items, and sets the impasse to 'NONE'.

```
__ PSInitialize ______________________________
   ΔDecide
   ___________
   ps_state' = PSInitialState
   items' = ∅
   impasse' = 'NONE'
______________________________________________
```

StartPS moves the state machine into the state to test for require preferences.

```
__ PSStart ___________________________________
   ΔDecide
   ___________
   ps_state = PSInitialState
   slot' = slot
   ps_state' = PSRequireState
______________________________________________
```

If there is exactly one require preference for the slot in memory and there is no prohibit preference for it in preference memory, then its value is the winner. The impasse is marked as `NONE', the value is returned in the singleton items set, and preference semantics is finished.

PSOneRequire

$\Delta Decide$

$ps\_state = PSRequireState$

$(\exists_1\ v : Symbol \bullet$
$\quad makeUnaryPreference(first(slot), second(slot), v, \text{`!'}) \in PM) \wedge$
$\neg\ (\exists\ v : Symbol \bullet$
$\quad makeUnaryPreference(first(slot), second(slot), v, \text{`!'}) \in PM \wedge$
$\quad makeUnaryPreference(first(slot), second(slot), v, \text{`\~{}'}) \in PM)$

$\exists_1\ v : Symbol \mid$
$\quad makeUnaryPreference(first(slot), second(slot), v, \text{`!'}) \in PM \bullet$
$\quad items' = \{v\}$

$impasse' = \text{`NONE'}$

$number\_of\_winners' = One$

$ps\_state' = PSFinishedState$

If there is more than one required candidate, then a constraint failure impasse is recognized with all of the required items in the items set, and preference semantics is complete.

PSMultipleRequires

$\Delta Decide$

$ps\_state = PSRequireState$

$\exists\ V : \mathbb{P}\ Symbol \mid$
$\quad (V = \{v : Symbol \mid$
$\qquad makeUnaryPreference(first(slot), second(slot), v, \text{`!'}) \in PM\} \wedge$
$\quad \# V > 1) \bullet$
$\quad items' = V$

$impasse' = \text{`CONSTRAINT-FAILURE'}$

$ps\_state' = PSFinishedState$

If there exists a required value that is also prohibited, a constraint impasse with the required/prohibited value is recognized and preference semantics terminates.

```
__ PSRequireProhibited ____________________________
  ΔDecide
  ──────────────
  $ps\_state = PSRequireState$

  $\exists v : Symbol \mid$
      $makeUnaryPreference(first(slot), second(slot), v, `!') \in PM \land$
      $makeUnaryPreference(first(slot), second(slot), v, `-') \in PM \bullet$
    $items' = \{v\}$

  $impasse' = `CONSTRAINT\text{-}FAILURE'$

  $ps\_state' = PSFinishedState$
────────────────────────────────────────────
```

If there are no require preferences, preference semantics continues on to consider acceptable preferences.

```
__ PSNoRequires ____________________________
  ΔDecide
  ──────────────
  $ps\_state = PSRequireState$

  $\neg (\exists v : Symbol \bullet$
    $makeUnaryPreference(first(slot), second(slot), v, `!') \in PM)$

  $impasse' = `NONE'$

  $items' = \varnothing$

  $ps\_state' = PSAcceptableState$
────────────────────────────────────────────
```

The require processing is stepped through by the *Require* transition. The *PSRequire* compound transition is defined to allow users a single place to hook all possible ways that preference semantics processes requires.

$$\begin{aligned} PSRequire \mathrel{\widehat{=}} {} & PSOneRequire \lor PSMultipleRequires \lor \\ & PSRequireProhibited \lor PSNoRequires \end{aligned}$$

The *PSAcceptable* transition collects all candidates with acceptable preferences in preference memory, stores them in the items set and moves the machine to the state that checks for prohibit preferences.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PSAcceptable}} \\
\Delta \mathit{Decide} \\
\hline
\mathit{ps\_state} = \mathit{PSAcceptableState} \\
\exists\, v : \mathit{Symbol} \bullet \\
\quad \mathit{makeUnaryPreference}(\mathit{first}(\mathit{slot}), \mathit{second}(\mathit{slot}), v, `+') \in \mathit{PM} \\
\mathit{slot}' = \mathit{slot} \\
\mathit{items}' = \{ v : \mathit{Symbol} \mid \\
\qquad \mathit{makeUnaryPreference}(\mathit{first}(\mathit{slot}), \mathit{second}(\mathit{slot}), v, `+') \in \mathit{PM} \} \\
\mathit{impasse}' = `\mathrm{NONE}' \\
\mathit{ps\_state}' = \mathit{PSProhibitState} \\
\hline
\end{array}
$$

The *PSNoAcceptable* transition causes preference semantics to exit with an empty items set when there are no acceptable preferences in memory for the slot. This transition is a compromise to the obvious implementation. Preference semantics would run exactly the same without it, but one of the common cases is re-deciding a slot that has had all of its acceptable preferences withdrawn and so every implementation would benefit from optimizing this case.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PSNoAcceptable}} \\
\Delta \mathit{Decide} \\
\hline
\mathit{ps\_state} = \mathit{PSAcceptableState} \\
\neg\, (\exists\, v : \mathit{Symbol} \bullet \\
\quad \mathit{makeUnaryPreference}(\mathit{first}(\mathit{slot}), \mathit{second}(\mathit{slot}), v, `+') \in \mathit{PM}) \\
\mathit{slot}' = \mathit{slot} \\
\mathit{items}' = \varnothing \\
\mathit{impasse}' = `\mathrm{NONE}' \\
\mathit{ps\_state}' = \mathit{PSFinishedState} \\
\hline
\end{array}
$$

The *PSProhibit* transition filters out of the current set of candidates those which have prohibit preferences, and moves the machine to a state that searches for reject preferences.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\, \mathit{PSProhibit}\,\rule{20em}{0.4pt}} \\
\Delta \mathit{Decide} \\
\hline
\mathit{ps\_state} = \mathit{PSProhibitState} \\
\mathit{slot}' = \mathit{slot} \\
\mathit{items}' = \{v : \mathit{items} \mid \\
\qquad \mathit{makeUnaryPreference}(\mathit{first}(\mathit{slot}), \mathit{second}(\mathit{slot}), v, \text{`\textasciitilde'}) \notin \mathit{PM}\} \\
\mathit{impasse}' = \text{`NONE'} \\
\mathit{ps\_state}' = \mathit{PSRejectState} \\
\hline
\end{array}
$$

The *PSReject* transition filters out of the set of items those that have reject preferences, and jumps to processing the better worse preferences.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\, \mathit{PSReject}\,\rule{20em}{0.4pt}} \\
\Delta \mathit{Decide} \\
\hline
\mathit{ps\_state} = \mathit{PSRejectState} \\
\mathit{slot}' = \mathit{slot} \\
\mathit{items}' = \{v : \mathit{items} \\
\qquad \mathit{makeUnaryPreference}(\mathit{first}(\mathit{slot}), \mathit{second}(\mathit{slot}), v, \text{`}-\text{'}) \notin \mathit{PM}\} \\
\mathit{impasse}' = \text{`NONE'} \\
\mathit{ps\_state}' = \mathit{PSBetterWorseState} \\
\hline
\end{array}
$$

The function *Conflict* is used to extract the candidates that have conflicting better/worse preferences.

$$
\begin{array}{l}
Conflict : \mathbb{P}\ Preference \times Slot \times \mathbb{P}\ Symbol \rightarrow \mathbb{P}\ Symbol \\
\hline
\forall PM : \mathbb{P}\ Preference;\ slot : Slot;\ V : \mathbb{P}\ Symbol \bullet \\
\quad Conflict(PM, slot, V) = \\
\quad\quad \{ j : V \mid \exists k : V \mid j \neq k \bullet \\
\quad\quad\quad (makeBinaryPreference(first(slot), second(slot), j, `>', k) \in PM \wedge \\
\quad\quad\quad\ makeBinaryPreference(first(slot), second(slot), k, `>', j) \in PM) \vee \\
\quad\quad\quad (makeBinaryPreference(first(slot), second(slot), j, `<', k) \in PM \wedge \\
\quad\quad\quad\ makeBinaryPreference(first(slot), second(slot), k, `<', j) \in PM) \vee \\
\quad\quad\quad (makeBinaryPreference(first(slot), second(slot), j, `>', k) \in PM \wedge \\
\quad\quad\quad\ makeBinaryPreference(first(slot), second(slot), j, `<', k) \in PM)\}
\end{array}
$$

If there are no conflicting better/worse preferences between the items, then *PSNoBetterWorseConflict* applies. It filters out of the items set the ones that have another item that is better, or are worse than another item. The state machine moves on to filter by best/worst preferences.

$$
\begin{array}{l}
\text{— } PSNoBetterWorseConflict \text{ ———————} \\
\Delta Decide \\
\hline
ps\_state = PSBetterWorseState \\
slot' = slot \\
Conflict(PM, slot, items) = \varnothing \\
items' = \{ v : items \mid \\
\quad\quad \neg (\exists w : items \bullet v \neq w \wedge \\
\quad\quad\quad makeBinaryPreference(first(slot), second(slot), v, `<', w) \in PM \wedge \\
\quad\quad\quad makeBinaryPreference(first(slot), second(slot), w, `>', v) \in PM)\} \\
impasse' = `\text{NONE}' \\
ps\_state' = PSBestState \\
\hline
\end{array}
$$

If there are better/worse conflicts, this transition sets the items augmentation to the a.' of the items involved in the conflicts, declares the impasse to be a conflict and finishes preference semantics.

$$
\begin{array}{|l}
\textit{PSBetterWorseConflict} \\
\hline
\Delta \textit{Decide} \\
\hline
\textit{ps\_state} = \textit{PSBetterWorseState} \\
\textit{slot}' = \textit{slot} \\
\textit{Conflict}(\textit{PM}, \textit{slot}, \textit{items}) \neq \emptyset \\
\textit{impasse} = \text{`CONFLICT'} \\
\textit{items}' = \textit{Conflict}(\textit{PM}, \textit{slot}, \textit{items}) \\
\textit{ps\_state}' = \textit{PSFinishedState} \\
\hline
\end{array}
$$

The processing of better/worse preferences occurs in either the *PSNoBetterWorseConflict* or the *PSBetterWorseConflict* transitions and is stepped by BetterWorse.

$$\textit{PSBetterWorse} \mathrel{\widehat{=}} \textit{PSNoBetterWorseConflict} \lor \textit{PSBetterWorseConflict}$$

The *PSBest* transition filters the candidates for those that have a best preference in memory. If there are no best preferences in memory for any of the current candidates, the entire old set of candidates is passed on. After best candidate filtering, preference semantics uses the worst preferences as a filter.

$$
\begin{array}{|l}
\textit{PSBest} \\
\hline
\Delta \textit{Decide} \\
\hline
\textit{ps\_state} = \textit{PSBestState} \\
\textit{slot}' = \textit{slot} \\
\exists\, V : \mathbb{P}\, \textit{Symbol} \mid \\
\quad V = \{v : \textit{items} \mid \\
\qquad \textit{makeUnaryPreference}(\textit{first}(\textit{slot}), \textit{second}(\textit{slot}), v, \text{`>'}) \in \textit{PM}\} \bullet \\
\quad (V \neq \emptyset \Rightarrow \textit{items}' = V \land \\
\quad V = \emptyset \Rightarrow \textit{items}' = \textit{items}) \\
\textit{impasse}' = \text{`NONE'} \\
\textit{ps\_state}' = \textit{PSWorstState} \\
\hline
\end{array}
$$

The *PSWorst* filter removes the items that have worst preferences in memory. If they all have worst preferences then none are filtered. After worst processing, preference semantics studies the indifferent preferences.

```
PSWorst
  $\Delta Decide$
  ----------
  $ps\_state = PSWorstState$
  $slot' = slot$
  $\exists V : \mathbb{P}\ Symbol \mid$
        $V = \{v : items \mid$
          $makeUnaryPreference(first(slot), second(slot), v, `<') \notin PM\} \bullet$
      $(V \neq \varnothing \Rightarrow items' = V \wedge$
       $V = \varnothing \Rightarrow items' = items)$
  $impasse' = `NONE'$
  $ps\_state' = PSIndifferentState$
```

The predicate *AllMutuallyIndifferent* checks that all of the candidates in the set are mutually indifferent. A candidate is mutually indifferent with the other candidates if it is unarily indifferent, or if there is a binary indifferent preference between the two candidates.

```
  $AllMutuallyIndifferent\_ : \mathbb{P}(\mathbb{P}\ Preference \times Slot \times \mathbb{P}\ Symbol)$
  ----------
  $\forall PM : \mathbb{P}\ Preference;\ slot : Slot;\ V : \mathbb{P}\ Symbol \bullet$
    $AllMutuallyIndifferent(PM, slot, V) \Leftrightarrow$
      $(\forall j, k : V \mid j \neq k \bullet$
        $(makeUnaryPreference(first(slot), second(slot), j, `=_P') \in PM \vee$
        $((makeBinaryPreference(first(slot), second(slot), j, `=_P', k) \in PM) \vee$
         $makeBinaryPreference(first(slot), second(slot), k, `=_P', j) \in PM)))$
```

If the candidates are all mutually indifferent, preference semantics returns with all of the candidates in the items set, an impasse of 'NONE' and sets the number of winners to one. This allows WMPhase and QPhase to tell that it has a set of indifferent candidates to choose one from.

PSMutuallyIndifferent

$\Delta Decide$

$ps\_state = PSIndifferentState$

$slot' = slot$

$impasse =$ 'NONE'

$AllMutuallyIndifferent(PM, slot, items)$

$items' = items$

$ps\_state' = PSFinishedState$

$number\_of\_winners' = One$

If the candidates are not mutually indifferent and the decision is not for a context slot, there is still a chance that all of the items are parallel. In this case. *PSNotMutuallyIndifferentNonContextSlot* passes the items on to the parallel checking state.

PSNotMutuallyIndifferentNonContextSlot

$\Delta Decide$

$ps\_state = PSIndifferentState$

$slot' = slot$

$impasse =$ 'NONE'

$\neg\ AllMutuallyIndifferent(PM, slot, items)$

$\neg\ (first(slot) \in GM \land$
$\quad second(slot) \in \{$'PROBLEM-SPACE', 'STATE', 'OPERATOR'$\})$

$items' = items$

$ps\_state' = PSParallelState$

Soar6 does not support parallel candidates for context items, although an earlier version of Soar did for operators. So non-mutually indifferent items for context slots generate a tie, with all of the candidates as items, not just those that are not mutually indifferent.

PSNotMutuallyIndifferentContextSlot

$\Delta Decide$

$ps\_state = PSIndifferentState$

$slot' = slot$

$impasse = \text{`NONE'}$

$\neg\ AllMutuallyIndifferent(PM, slot, items)$

$first(slot) \in GM \land$
$second(slot) \in \{\text{`PROBLEM-SPACE'}, \text{`STATE'}, \text{`OPERATOR'}\}$

$items' = items$

$ps\_state' = PSFinishedState$

$impasse' = \text{`TIE'}$

The complete indifference processing is handled by one of the three transitions above, and stepped and hooked using the *Indifferent* transition.

$$\begin{aligned} PSIndifferent \mathrel{\widehat=}\ & PSMutuallyIndifferent \lor \\ & PSNotMutuallyIndifferentNonContextSlot \lor \\ & PSNotMutuallyIndifferentContextSlot \end{aligned}$$

The parallel state uses the predicate *AllMutuallyParallel* to check if the set of candidates are all mutually parallel. An item is mutually parallel to the set of other items if it has a unary parallel preference, or if it has either of the two possible relative parallels between each item.

$AllMutuallyParallel\_ : \mathbb{P}(\mathbb{P}\ Preference \times Slot \times \mathbb{P}\ Symbol)$

$$\begin{aligned} & \forall\, PM : \mathbb{P}\ Preference;\ slot : Slot;\ V : \mathbb{P}\ Symbol \bullet \\ & \quad AllMutuallyParallel(PM, slot, V) \Leftrightarrow \\ & \quad\quad (\forall\, j, k : V \mid j \neq k \bullet \\ & \quad\quad\quad (makeUnaryPreference(first(slot), second(slot), j, \text{`\&'}) \in PM \lor \\ & \quad\quad\quad\ makeBinaryPreference(first(slot), second(slot), j, \text{`\&'}, k) \in PM \lor \\ & \quad\quad\quad\ makeBinaryPreference(first(slot), second(slot), k, \text{`\&'}, j) \in PM)) \end{aligned}$$

If all of the candidates are mutually parallel *PSMutuallyParallel* returns all of the items, sets impasse to `NONE´ and the number of winners to All.

PSMutuallyParallel

$$\Delta Decide$$

$$\begin{array}{l} ps\_state = PSParallelState \\ slot' = slot \\ AllMutuallyParallel(PM, slot, items) \\ items' = items \\ impasse' = \text{`NONE´} \\ number\_of\_winners' = All \\ ps\_state' = PSFinishedState \end{array}$$

PSNotMutuallyParallel returns a tie impasse if all of the candidates are not mutually parallel.

PSNotMutuallyParallel

$$\Delta Decide$$

$$\begin{array}{l} ps\_state = PSParallelState \\ slot' = slot \\ \neg\ AllMutuallyParallel(PM, slot, items) \\ items' = items \\ impasse' = \text{`TIE´} \\ ps\_state' = PSFinishedState \end{array}$$

The *PSParallel* transition is defined to either be a *MutuallyParallel* transition or a *NotMutuallyParallel* transition.

$$PSParallel \mathrel{\widehat{=}} PSMutuallyParallel \wedge PSNotMutuallyParallel$$

The operation to step preference semantics executes the start transition, or one of the transitions associated with the processing for a specific preference.

$$\begin{array}{l} PSStep \mathrel{\widehat{=}} PSStart \vee \\ \quad PSRequire \vee PSAcceptable \vee PSNoAcceptable \vee PSProhibit \vee PSReject \vee \\ \quad PSBetterWorse \vee PSBest \vee PSWorst \vee PSIndifferent \vee PSParallel \end{array}$$

7.9 Working Memory Phase Transitions

Working Memory Phase is defined in 12 transitions (Figure 7.3).

1. *WMPhaseInitialize* — prepares the WMPhase state machine for execution.
2. *WMPhaseReset* — resets the state of WMPhase when its execution is terminated abnormally.
3. *WMPhaseStartPickSlot* — starts the loop over the changed slots.
4. *WMPhaseStartDecideSlot* — selects one of the changed slots to decide.
5. *WMPhaseStepDecideSlot* — runs the preference semantics on the chosen slot.
6. *WMPhaseStartImpasser* — starts the execution of the impasser on the slot.
7. *WMPhaseStepImpasser* — steps the impasser state machine.
8. *WMPhaseImpasserFinished* — when the impasser is finished, this moves the machine on to the WMPhaseChangeWMState which adds and removes working memory elements for the slot.
9. *WMPhaseRemoveWME* — this removes any working memory elements no longer supported by the slot's decision.
10. *WMPhaseAddWME* — this adds in any working memory elements that are required by the slot's decision.
11. *WMPhaseChangeWMFinished* — when all of the required changes to working memory have been accomplished, this returns control to selecting another changed slot.
12. *WMPhaseFinish* — when all of the changed non-context slots have been decided, this stops the execution of WMPhase.

WMPhaseInitialize initializes the WMPhase by resetting the state counter.

```
WMPhaseInitialize
  ΔDecide
  ─────────
  wmphase_state = WMPhaseFinishedState
  wmphase_state' = WMPhaseInitialState
```

WMPhaseReset resets the system if WMPhase's operation is abnormally terminated from the external interface.

$$\begin{array}{|l}
\textit{WMPhaseReset} \\
\hline
\Delta \textit{Decide} \\
\hline
\textit{wmphase\_state}' = \textit{WMPhaseInitialState} \\
\hline
\end{array}$$

WMPhaseStartPickSlot moves WMPhase into the state that will select a changed slot to decide.

$$\begin{array}{|l}
\textit{WMPhaseStartPickSlot} \\
\hline
\Delta \textit{Decide} \\
\hline
\textit{wmphase\_state} = \textit{WMPhaseInitialState} \\
\textit{wmphase\_state}' = \textit{WMPhasePickSlotState} \\
\hline
\end{array}$$

The recognition memory records all of the slots that had preference changes during the last preference phase in the set *changed_slots*. In the pick slot state WMPhase picks one of these slots to have its preferences decided, initializes the preference semantics and moves to the state that decides the slot.

$$\begin{array}{|l}
\textit{WMPhaseStartDecideSlot} \\
\hline
\Delta \textit{Decide} \\
\hline
\textit{wmphase\_state} = \textit{WMPhasePickSlotState} \\
\exists\, s : \textit{changed\_slots} \setminus (\textit{GM} \times \textit{ContextSlotAttribute}) \bullet \\
\quad (\textit{changed\_slots}' = \textit{changed\_slots} \setminus \{s\} \wedge \\
\quad \textit{slot}' = s) \\
\textit{PSInitialize} \\
\textit{slot}' = \textit{slot} \\
\textit{wmphase\_state}' = \textit{WMPhaseDecideSlotState} \\
\hline
\end{array}$$

The *WMPhaseStepDecideSlot* transition steps preference semantics until a decision has been reached.

WMPhaseStepDecideSlot

$\Delta Decide$

$wmphase\_state = WMPhaseDecideSlotState = wmphase\_state'$

$PSStep$

$slot' = slot$

When a decision has been reached preference semantic's precondition is no longer satisfied, so *WMPhaseStartImpasser* initializes the impasser and moves WMPhase to the *WMPhaseImpasserState* state to run the impasser.

WMPhaseStartImpasser

$\Delta Decide$

$wmphase\_state = WMPhaseDecideSlotState$

$\neg \text{ pre } PSStep$

$ImpasserInitialize$

$slot' = slot$

$wmphase\_state' = WMPhaseImpasserState$

The *WMPhaseStepImpasser* transition runs the impasser. The impasser looks at the decision that preference semantics has created, and changes working memory and impasse memory to match. If the decision requires an impasse, the impasser will create it. If the decision requires a change to the existing impasse, the impasser will change it. If the decision requires no impasse, the impasser will remove any existing impasse. If the decision requires no impasse and none exists in memory, the impasser will make no changes.

WMPhaseStepImpasser

$\Delta Decide$

$wmphase\_state = WMPhaseImpasserState = wmphase\_state'$

$ImpasserStep$

$slot' = slot$

The *WMPhaseImpasserFinished* transition recognizes when the impasser has finished by checking its precondition. When it has completed, it moves the WMPhase on to the state that changes the values in working memory to match the slot's decision.

WMPhaseImpasserFinished

$\Delta Decide$

$wmphase\_state = WMPhaseImpasserState$

$\neg\ \mathrm{pre}\ ImpasserStep$

$slot' = slot$

$wmphase\_state' = WMPhaseChangeWMState$

If there is no impasse, and there is not some element in working memory that the decision requires then the *WMPhaseAddWME* transition will add it. If the decision is for a set of all parallel items, then *WMPhaseAddWME* will add them all, one at a time. If the decision is for one of a set of indifferent candidates and one candidate is not already installed, then *WMPhaseAddWME* will pick one of the items non-deterministically and add it to memory. Chunking's *TraceWME* operation is referenced to extend the instance numbering and the working memory trace.

$$\begin{array}{|l}
\textit{WMPhaseAddWME} \\
\hline
\Delta \textit{Decide} \\
\hline
\textit{wmphase\_state} = \textit{WMPhaseChangeWMState} = \textit{wmphase\_state}' \\
\textit{slot}' = \textit{slot} \\
\textit{items}' = \textit{items} \\
\textit{impasse} = \text{`NONE'} \\
\exists\, v : \textit{items} \mid \\
\quad (\textit{number\_of\_winners} = \textit{All} \land \\
\quad \neg\, (\exists\, w : \textit{WM} \bullet \\
\quad\quad w.\textit{id} = \textit{first}(\textit{slot}) \land w.\textit{attribute} = \textit{second}(\textit{slot}) \land \\
\quad\quad w.\textit{value} = v \land w.\textit{context\_acceptable\_preference} = \textit{No})) \lor \\
\quad (\textit{number\_of\_winners} = \textit{One} \land \\
\quad \neg\, (\exists\, w : \textit{WM} \bullet \\
\quad\quad w.\textit{id} = \textit{first}(\textit{slot}) \land w.\textit{attribute} = \textit{second}(\textit{slot}) \land \\
\quad\quad w.\textit{value} \in \textit{items} \land w.\textit{context\_acceptable\_preference} = \textit{No})) \bullet \\
\;\; \exists\, w : \textit{WME} \mid w = \textit{makeWME}(\textit{first}(\textit{slot}), \textit{second}(\textit{slot}), v, \textit{No}) \land \\
\quad \textit{WM}' = \textit{WM} \cup \{w\} \land \\
\quad \textit{TraceWME}) \\
\hline
\end{array}$$

If there is an element in working memory that does not match the decision, the *WMPhaseRemoveWME* transition removes it. If the decision resulted in an impasse, then *WMPhaseRemoveWME* removes all of the working memory elements for the slot. If the decision resulted in no impasse, and there is a value in memory that is not in the items set, then it is removed. If the decision is for a set of mutually indifferent candidates and more than one of them is in memory, then all but one are chosen non-deterministically to removed.

WMPhaseRemoveWME

$\Delta Decide$

$$wmphase\_state = WMPhaseChangeWMState = wmphase\_state'$$
$$slot' = slot$$
$$items' = items$$
$$\begin{array}{l} \exists\, w : WM \mid \\ \quad w.id = first(slot) \land w.attribute = second(slot) \land \\ \quad w.context\_acceptable\_preference = No \land \\ \quad ((impasse \neq \text{`NONE'}) \lor \\ \quad (w.value \notin items) \lor \\ \quad (number\_of\_winners = One \land \\ \quad (\exists\, w_2 : WM \bullet \\ \qquad w \neq w_2 \land w_2.id = first(slot) \land \\ \qquad w_2.attribute = second(slot) \land \\ \qquad w_2.context\_acceptable\_preference = No \land \\ \qquad w_2.value \in items))) \bullet \\ \quad WM' = WM \setminus \{w\} \end{array}$$

When no more elements need to be added to or removed from working memory, then the *WMPhaseChangeWMFinished* transition moves WMPhase back to the decide slot state.

WMPhaseChangeWMFinished

$\Delta Decide$

$$wmphase\_state = WMPhaseChangeWMState$$
$$\neg\,(\text{pre } WMPhaseAddWME \lor \text{pre } WMPhaseRemoveWME)$$
$$wmphase\_state' = WMPhasePickSlotState$$

When all of the changed slots have been decided, *WMPhaseFinish* finishes WMPhase.

$$\begin{array}{|l}
\hline
\textit{WMPhaseFinish} \\
\Delta \textit{Decide} \\
\hline
\textit{wmphase\_state} = \textit{WMPhasePickSlotState} \\
\textit{changed\_slots} \setminus (\textit{GM} \times \textit{ContextSlotAttribute}) = \emptyset \\
\textit{wmphase\_state}' = \textit{WMPhaseFinishedState} \\
\hline
\end{array}$$

The WMPhase is stepped by taking any one of its transitions, with the exception of the initializing transition.

$$\begin{aligned}
\textit{WMPhaseStep} \mathrel{\widehat{=}} {} & \textit{WMPhaseStartPickSlot} \lor \textit{WMPhaseStartDecideSlot} \lor \\
& \textit{WMPhaseStepDecideSlot} \lor \textit{WMPhaseStartImpasser} \lor \\
& \textit{WMPhaseStepImpasser} \lor \textit{WMPhaseImpasserFinished} \lor \\
& \textit{WMPhaseAddWME} \lor \textit{WMPhaseRemoveWME} \lor \\
& \textit{WMPhaseChangeWMFinished} \lor \textit{WMPhaseFinish}
\end{aligned}$$

7.10 Quiescence Phase Transitions

QPhase is very parallel in structure to WMPhase, but differs because it must walk the context stack instead of simply processing the changed slots. It adds in four transitions to the basic structure of working memory phase, for a total of 16 transitions (see Figure 7.4).

1. *QPhaseInitialize* — initializes QPhase.
2. *QPhaseReset* — resets QPhase's state when its operation is abnormally terminated.
3. *QPhaseStartPickSlot* — starts QPhase iterating through the slots of the context stack from oldest to newest.
4. *QPhaseNextSlot* — if the slot under consideration does not need to be decided, this transition picks the next slot in the context stack to try.
5. *QPhaseNoChangedPreferencesNoChangeImpasse* — if the slot under consideration has no changed preferences and is the last slot of the context stack, this transition creates a no-change impasse.
6. *QPhaseStartDecideSlot* — this starts QPhase running the preference semantics.
7. *QPhaseStepDecideSlot* — this transition steps preference semantics.

8. *QPhaseStartImpasser* — when preference semantics is finished, this transition moves QPhase on to the state that handles impassing.

9. *QPhaseChangedPreferencesNoChangeImpasse* — if the slot is the last slot of the context and has changed preferences but the preference semantics does not produce an impasser or a winning candidate, then this transition creates a no-change impasse.

10. *QPhaseStepImpasser* — steps the impasser on the current slot.

11. *QPhaseImpasserFinished* — moves the QPhase from stepping the impasser into the state that changes working memory.

12. *QPhaseAddWME* — this transition adds the working memory elements for slots that preference semantics specifies.

13. *QPhaseRemoveWME* — this transition removes the working memory elements for the slot that preference semantics no longer specifies.

14. *QPhaseReAddWME* — this transition re-adds a working memory element for a slot that has been explicitly reconsidered.

15. *QPhaseSlotUnchanged* — if QPhase has not changed the slot under consideration then this loops back to the pick slot state to decide another slot.

16. *QPhaseSlotChanged* — if QPhase has changed the slot, then this ends the execution of QPhase.

QPhaseInitialize initializes the QPhase state machine by setting its state counter to the initial state.

QPhaseInitialize

$\Delta Decide$

$qphase\_state' = QPhaseFinishedState$

$qphase\_state' = QPhaseInitialState$

QPhaseReset resets the QPhase's state when the machine is terminated abnormally.

QPhaseReset

$\Delta Decide$

$qphase\_state' = QPhaseInitialState$

$changed\_slots = \emptyset$

QPhase iterates over the slots of the context from oldest slot to newest slot. *QPhaseStartPickSlot* starts the loop by setting the initial slot to the top goal and the problem-space attribute, and entering the *QPhasePickSlotState*.

$$
\begin{array}{l}
\textit{QPhaseStartPickSlot} \\
\hline
\Delta \textit{Decide} \\
\hline
\textit{qphase\_state} = \textit{QPhaseInitialState} \\
\textit{slot} = (\textit{top\_goal}, \text{`PROBLEM-SPACE'}) \\
\textit{qphase\_state}' = \textit{QPhasePickSlotState} \\
\hline
\end{array}
$$

NextContextSlot orders the slots to allow QPhase to know which slot of the goal to consider next. The relation's minimum is 'GOAL' to simplify the calculation of the slot of no-change impasses.

$$
\begin{array}{l}
\textit{NextContextSlot} : \textit{ContextSlotAttribute} \twoheadrightarrow \textit{ContextSlotAttribute} \\
\hline
\textit{NextContextSlot} = \{(\text{`GOAL'}, \text{`PROBLEM-SPACE'}), \\
\qquad\qquad (\text{`PROBLEM-SPACE'}, \text{`STATE'}), \\
\qquad\qquad (\text{`STATE'}, \text{`OPERATOR'})\}
\end{array}
$$

QPhase must know when the slot it is deciding is the last one that it should consider for this impasse. If the slot is the operator, it is the last to check as it is the last in the impasse. If the context does not have a working memory element in memory for the slot, then it is the last one to check, e.g., you can't install an operator unless there is a state.

$$
\begin{array}{l}
\textit{LastSlotToCheckInContext\_} : \mathbb{P}(\textit{Slot} \times \mathbb{P}\ \textit{WME}) \\
\hline
\forall\, s : \textit{Slot};\ \textit{WM} : \mathbb{P}\ \textit{WME} \bullet \\
\quad \textit{LastSlotToCheckInContext}(s, \textit{WM}) \Leftrightarrow \\
\qquad (\textit{second}(s) = \text{`OPERATOR'} \lor \\
\qquad \neg\, (\exists\, w : \textit{WM} \bullet \\
\qquad\quad w.\textit{id} = \textit{first}(s) \land w.\textit{attribute} = \textit{second}(s) \land \\
\qquad\qquad w.\textit{context\_acceptable\_preference} = \textit{No}))
\end{array}
$$

QPhase uses the *LastSlotToCheckInContextStack* predicate to determine the last one of the context stack for it to process. A slot is the last one to check in the stack if it is the last slot to check in the last impasse. It will signal a no-change impasse for the last filled slot if preference semantics can not reach a conclusion about this slot.

$$
\begin{array}{|l}
\mathit{LastSlotToCheckInContextStack\_} : \mathbb{P}(\mathit{Slot} \times \mathbb{P}\ \mathit{Identifier} \times \mathbb{P}\ \mathit{WME}) \\
\hline
\forall\, s : \mathit{Slot};\ \mathit{GM} : \mathbb{P}\ \mathit{Identifier};\ \mathit{WM} : \mathbb{P}\ \mathit{WME} \bullet \\
\quad \mathit{LastSlotToCheckInContextStack}(s, \mathit{GM}, \mathit{WM}) \Leftrightarrow \\
\quad\quad \neg\, (\exists\, g : \mathit{GM} \bullet \mathit{makeWME}(g, \text{`OBJECT'}, \mathit{first}(s), \mathit{No}) \in \mathit{WM}) \wedge \\
\quad\quad \mathit{LastSlotToCheckInContext}(s, \mathit{WM})
\end{array}
$$

QPhase must decide a context slot if there is no value for it in working memory and there are preference changes, or if there is a value in working memory and there is a reconsider.

$$
\begin{array}{|l}
\mathit{DecidableSlot\_} : \mathbb{P}(\mathit{Slot} \times \mathbb{P}\ \mathit{WME} \times \mathbb{P}\ \mathit{Preference} \times \mathbb{P}\ \mathit{Slot}) \\
\hline
\forall\, \mathit{slot} : \mathit{Slot};\ \mathit{WM} : \mathbb{P}\ \mathit{WME};\ \mathit{PM} : \mathbb{P}\ \mathit{Preference};\ \mathit{changed\_slots} : \mathbb{P}\ \mathit{Slot} \bullet \\
\quad \mathit{DecidableSlot}(\mathit{slot}, \mathit{WM}, \mathit{PM}, \mathit{changed\_slots}) \Leftrightarrow \\
\quad\quad (\neg\, (\exists\, v : \mathit{Symbol} \bullet \\
\quad\quad\quad \mathit{makeWME}(\mathit{first}(\mathit{slot}), \mathit{second}(\mathit{slot}), v, \mathit{No}) \in \mathit{WM}) \wedge \\
\quad\quad\quad \mathit{slot} \in \mathit{changed\_slots}) \vee \\
\quad\quad (\exists\, v : \mathit{Symbol} \bullet \\
\quad\quad\quad \mathit{makeWME}(\mathit{first}(\mathit{slot}), \mathit{second}(\mathit{slot}), v, \mathit{No}) \in \mathit{WM} \wedge \\
\quad\quad\quad \mathit{makeUnaryPreference}(\mathit{first}(\mathit{slot}), \mathit{second}(\mathit{slot}), v, \text{`@'}) \in \mathit{PM})
\end{array}
$$

While QPhase is iterating across the context stack, *QPhaseNextSlot* allows QPhase to skip slots that do not need to be decided. If the slot that QPhase is working with is not decidable and it is not the last slot to check, then it picks the next slot to decide. If the slot is not the last in this context to check, it moves to the next slot in the same context. If the slot is the last in this context to decide, it starts on the problem space of the next context.

$$\begin{array}{|l}
\textit{QPhaseNextSlot} \\
\hline
\Delta \textit{Decide} \\
\hline
\textit{qphase\_state} = \textit{QPhasePickSlotState} = \textit{qphase\_state}' \\
\neg\ \textit{DecidableSlot}(\textit{slot}, \textit{WM}, \textit{PM}, \textit{changed\_slots}) \\
\neg\ \textit{LastSlotToCheckInContextStack}(\textit{slot}, \textit{GM}, \textit{WM}) \\
\neg\ \textit{LastSlotToCheckInContext}(\textit{slot}, \textit{WM}) \Rightarrow \\
\quad \textit{slot}' = (\textit{first}(\textit{slot}), \textit{NextContextSlot}(\textit{second}(\textit{slot}))) \\
\textit{LastSlotToCheckInContext}(\textit{slot}, \textit{WM}) \Rightarrow \\
\quad (\exists\, \textit{sg} : \textit{GM} \mid \textit{makeWME}(\textit{sg}, \text{`OBJECT'}, \textit{first}(\textit{slot}), \textit{No}) \in \textit{WM} \bullet \\
\qquad \textit{slot}' = (\textit{sg}, \text{`PROBLEM-SPACE'})) \\
\hline
\end{array}$$

If QPhase's loop down the context stack takes it to the last slot in the context, and this slot has no preference changes, this transition signals a no-change impasse. The no change impasse is for the last filled slot of the context, or if the context has no filled slot it is a goal no-change.

QPhaseNoChangedPreferencesNoChangeImpasse

$\Delta Decide$

$qphase\_state = QPhasePickSlotState$

$\neg\ DecidableSlot(slot, WM, PM, changed\_slots)$

$LastSlotToCheckInContextStack(slot, GM, WM)$

$$\neg\,(\exists\, w : WM \bullet w.id = first(slot) \wedge w.attribute = second(slot) \wedge w.context\_acceptable\_preference = No) \Rightarrow slot' = (first(slot), NextContextSlot^{\sim}(second(slot)))$$

$$(\exists\, w : WM \bullet w.id = first(slot) \wedge w.attribute = second(slot) \wedge w.context\_acceptable\_preference = No) \Rightarrow slot' = (first(slot), second(slot))$$

$ImpasserInitialize$

$impasse' =$ 'NO-CHANGE'

$items' = \emptyset$

$qphase\_state' = QPhaseImpasserState$

If there are changed preferences for a slot, QPhase moves to the *QPhaseDecideSlotState* to step preference semantics. The slot to be be decided is removed from the changed_slots set so that it is not decided again.

QPhaseStartDecideSlot

$\Delta Decide$

$qphase\_state = QPhasePickSlotState$

$DecidableSlot(slot, WM, PM, changed\_slots)$

$changed\_slots' = changed\_slots \setminus \{slot\}$

$PSInitialize$

$context\_changed = Unchanged$

$qphase\_state' = QPhaseDecideSlotState$

QPhaseStepDecideSlot steps preference semantics on the current slot.

$$\begin{array}{|l}
\textit{QPhaseStepDecideSlot} \\
\Delta \textit{Decide} \\
\hline
\textit{qphase\_state} = \textit{QPhaseDecideSlotState} = \textit{qphase\_state}' \\
\textit{PSStep} \\
\hline
\end{array}$$

When preference semantics has finished, this transition moves QPhase to the *QPhaseImpasserState* to step the impasser. The impasser will add an impasse, remove an old impasse, change the items of the existing impasse, or do nothing depending upon the contents of working memory and the results of preference semantics.

$$\begin{array}{|l}
\textit{QPhaseStartImpasser} \\
\Delta \textit{Decide} \\
\hline
\textit{qphase\_state} = \textit{QPhaseDecideSlotState} \\
\neg \text{ pre } \textit{PSStep} \\
\neg\ \textit{LastSlotToCheckInContextStack}(\textit{slot}, \textit{GM}, \textit{WM}) \lor \\
\textit{impasse} \neq \text{`NONE'} \lor \\
\textit{items} \neq \emptyset \\
\textit{ImpasserInitialize} \\
\textit{qphase\_state}' = \textit{QPhaseImpasserState} \\
\hline
\end{array}$$

If the slot under consideration is the last slot in the context and preference semantics has no winner, then *QPhaseChangedPreferencesNoChangeImpasse* sets the impasse to no-change before starting the impasser.

$$
\begin{array}{|l}
\textit{QPhaseChangedPreferencesNoChangeImpasse} \\
\Delta \textit{Decide} \\
\hline
\textit{qphase\_state} = \textit{QPhaseDecideSlotState} \\
\neg \ \text{pre } \textit{PSStep} \\
\textit{items} = \varnothing \\
\textit{impasse} = \text{`NONE'} \\
\textit{LastSlotToCheckInContextStack}(\textit{slot}, \textit{GM}, \textit{WM}) \\
\neg (\exists w : \textit{WM} \bullet w.\textit{id} = \textit{first}(\textit{slot}) \wedge w.\textit{attribute} = \textit{second}(\textit{slot}) \wedge \\
\qquad w.\textit{context\_acceptable\_preference} = \textit{No}) \Rightarrow \\
\quad \textit{slot}' = (\textit{first}(\textit{slot}), \textit{NextContextSlot}^{\sim}(\textit{second}(\textit{slot}))) \\
(\exists w : \textit{WM} \bullet w.\textit{id} = \textit{first}(\textit{slot}) \wedge w.\textit{attribute} = \textit{second}(\textit{slot}) \wedge \\
\qquad w.\textit{context\_acceptable\_preference} = \textit{No}) \Rightarrow \\
\quad \textit{slot}' = (\textit{first}(\textit{slot}), \textit{second}(\textit{slot})) \\
\textit{ImpasserInitialize} \\
\textit{qphase\_state}' = \textit{QPhaseImpasserState} \\
\hline
\end{array}
$$

QPhaseStepImpasser steps the impasser state machine.

$$
\begin{array}{|l}
\textit{QPhaseStepImpasser} \\
\Delta \textit{Decide} \\
\hline
\textit{qphase\_state} = \textit{QPhaseImpasserState} = \textit{qphase\_state}' \\
\textit{ImpasserStep} \\
\hline
\end{array}
$$

When the impasser is finished, *QPhaseImpasserFinished* moves QPhase on to *ChangeWMState* to add and remove working memory elements for the slot.

$$
\begin{array}{|l}
\textit{QPhaseImpasserFinished} \\
\Delta \textit{Decide} \\
\hline
\textit{qphase\_state} = \textit{QPhaseImpasserState} \\
\neg \ \text{pre } \textit{ImpasserStep} \\
\textit{qphase\_state}' = \textit{QPhaseChangeWMState} \\
\hline
\end{array}
$$

Whenever QPhase changes a context slot, it must clear out all of the working memory elements for the lower slots of the same context. *LaterSlotsOfContextSlot* returns the working memory elements for the later slots so that they can be removed from working memory.

$$
\begin{array}{l}
\mathit{LaterSlotsOfContextSlot} : \mathit{Slot} \times \mathbb{P}\ \mathit{WME} \to \mathbb{P}\ \mathit{WME} \\
\hline
\forall\, s : \mathit{Slot};\ \mathit{WM} : \mathbb{P}\ \mathit{WME} \bullet \\
\quad \mathit{LaterSlotsOfContextSlot}(s, \mathit{WM}) = \\
\qquad \{ w : \mathit{WM} \mid w.\mathit{id} = \mathit{first}(s) \land w.\mathit{context\_acceptable\_preference} = \mathit{No} \land \\
\qquad\quad ((\mathit{second}(s) = \text{`PROBLEM-SPACE'} \land \\
\qquad\quad\ w.\mathit{attribute} \in \{\text{`STATE'}, \text{`OPERATOR'}\}) \lor \\
\qquad\quad (\mathit{second}(s) = \text{`STATE'} \land w.\mathit{attribute} = \text{`OPERATOR'}))\}
\end{array}
$$

If the preference semantics produces a set of items and no impasse, and none of the items is in working memory then *QPhaseAddWME* adds a working memory element for one of the items. The item used is picked non-deterministically. It marks the context as changed so that QPhase knows that it has found a change and that it should stop iterating across the slots. As it does this, it also deletes all of the later slots for the same context.

$$
\begin{array}{l}
\text{— } \mathit{QPhaseAddWME} \text{ ——————} \\
\Delta \mathit{Decide} \\
\hline
\mathit{qphase\_state} = \mathit{QPhaseChangeWMState} = \mathit{qphase\_state}' \\
\mathit{impasse} = \text{`NONE'} \\
\neg\, (\exists\, w : \mathit{WM} \bullet w.\mathit{id} = \mathit{first}(\mathit{slot}) \land w.\mathit{attribute} = \mathit{second}(\mathit{slot}) \land \\
\qquad w.\mathit{context\_acceptable\_preference} = \mathit{No} \land w.\mathit{value} \in \mathit{items}) \\
\mathit{context\_changed} = \mathit{Unchanged} \Rightarrow \mathit{context\_changed}' = \mathit{Changed} \\
\exists\, i : \mathit{items};\ w : \mathit{WME} \mid w = \mathit{makeWME}(\mathit{first}(\mathit{slot}), \mathit{second}(\mathit{slot}), i, \mathit{No}) \bullet \\
\qquad (\mathit{GM}' = \mathit{GM} \setminus \mathit{GoalDescendants}(\mathit{first}(\mathit{slot}), \mathit{GM}, \mathit{WM}) \land \\
\quad\ \mathit{WM}' = \mathit{WM} \setminus \mathit{LaterSlotsOfContextSlot}(\mathit{slot}, \mathit{WM}) \\
\qquad\quad \cup \{w\} \land \\
\qquad\quad \mathit{TraceWME}) \\
\hline
\end{array}
$$

QPhaseRemoveWME deletes any elements for the context slot that are not in the items set decided by preference semantics. It marks the context changed and deletes the elements for later slots. The one exception to this rule is that context slots are not emptied for no-change impasses. It also updates the bottom_goal pointer, in case it has deleted context impasses.

QPhaseRemoveWME

$\Delta Decide$

$$qphase\_state = QPhaseChangeWMState = qphase\_state'$$

$$context\_changed = Unchanged \Rightarrow context\_changed' = Changed$$

$$\begin{aligned}&\exists\, w : WM \mid \\ &\quad w.id = first(slot) \wedge w.attribute = second(slot) \wedge \\ &\quad w.context\_acceptable\_preference = No \wedge \\ &\quad w.value \notin items \bullet \\ &\quad WM' = WM \setminus \{r : WM \mid r = w \wedge impasse \neq \text{`NO-CHANGE'}\} \\ &\quad\quad \setminus LaterSlotsOfContextSlot(slot, WM)\end{aligned}$$

$$bottom\_goal' = first(slot)$$

Sometimes, QPhase re-decides an installed slot because it has a reconsider preference and preference semantics produces no impasse and an items set that holds the installed value. *QPhaseReAddWME* re-installs this candidate by removing the working memory element, the lower context slots, all subgoals and their architecturally created elements, and then re-adds the working memory element.

QPhaseReAddWME

$$\Delta Decide$$

$$\begin{array}{l}
qphase\_state = QPhaseChangeWMState \\[6pt]
context\_changed' = Changed \\[6pt]
impasse = \text{`NONE'} \\[6pt]
\exists\, w : WM \mid \\
\quad w.id = first(slot) \wedge w.attribute = second(slot) \wedge \\
\quad w.context\_acceptable\_preference = No \wedge \\
\quad w.value \in items \wedge \\
\quad makeUnaryPreference(first(slot), second(slot), w.value, \text{`@'}) \in PM \bullet \\
\;\; (GM' = GM \setminus GoalDescendants(first(slot), GM, WM) \wedge \\
\;\;\; WM' = WM \setminus \{w\} \\
\;\;\;\; \setminus LaterSlotsOfContextSlot(slot, WM) \\
\;\;\;\; \setminus (\bigcup\{g : GoalDescendants(first(slot), GM, WM) \bullet \\
\qquad\qquad ImpassesWMEs(g, GM, WM)\}) \\
\qquad\qquad \cup \{w\} \wedge \\
\qquad\qquad TraceWME) \\[6pt]
qphase\_state' = QPhaseFinishedState
\end{array}$$

If working memory is already consistent with the decision for the slot then *QPhaseSlotUnchanged* will recognize that the context is unchanged and move QPhase back into the pick slot state to continue the loop over the slots.

QPhaseSlotUnchanged

$$\Delta Decide$$

$$\begin{array}{l}
qphase\_state = QPhaseChangeWMState \\[6pt]
\neg\, (\text{pre } QPhaseAddWME \vee \text{pre } QPhaseRemoveWME) \\[6pt]
context\_changed = Unchanged \\[6pt]
qphase\_state' = QPhasePickSlotState
\end{array}$$

If neither the Impasser or the *ChangeWMState* have changed memory and all adds and removes have occurred, then *QPhaseSlotChanged* finishes the execution of QPhase.

$$
\begin{array}{|l}
\hline
\textit{QPhaseSlotChanged} \\
\Delta \textit{Decide} \\
\hline
\textit{qphase\_state} = \textit{QPhaseChangeWMState} \\
\neg\ (\text{pre } \textit{QPhaseAddWME} \lor \text{pre } \textit{QPhaseRemoveWME}) \\
\textit{context\_changed} = \textit{Changed} \\
\textit{qphase\_state}' = \textit{QPhaseFinishedState} \\
\hline
\end{array}
$$

QPhaseStep drives the QPhase one step forward in the state machine by executing one of the QPhase state transitions.

$$
\begin{aligned}
\textit{QPhaseStep} \mathrel{\widehat=} {} & \textit{QPhaseStartPickSlot} \lor \textit{QPhaseNextSlot} \lor \\
& \textit{QPhaseNoChangedPreferencesNoChangeImpasse} \lor \\
& \textit{QPhaseStartDecideSlot} \lor \textit{QPhaseStepDecideSlot} \lor \\
& \textit{QPhaseStartImpasser} \lor \textit{QPhaseStepImpasser} \lor \\
& \textit{QPhaseChangedPreferencesNoChangeImpasse} \lor \\
& \textit{QPhaseImpasserFinished} \lor \\
& \textit{QPhaseRemoveWME} \lor \textit{QPhaseAddWME} \lor \textit{QPhaseReAddWME} \lor \\
& \textit{QPhaseSlotUnchanged} \lor \textit{QPhaseSlotChanged}
\end{aligned}
$$

Chapter 8

Top Level

The actions of Soar's modules are sequenced by a top level state machine that steps the various control points of each module. Recognition memory is stepped through preference phase. Decide is stepped through WMPhase and QPhase. IO is stepped through the input cycle and the output cycle.

This chapter specifies.the states of the top level state machine, Soar's state schema and the transitions of the top level state machine.

8.1 The States of the Top Level

Soar's top level state machine has ten states (Figure 8.1).

1. *TLInitialState* — the machine starts here when initialized.
2. *TLDecisionCycleState* — the top level enters this state before starting each decision cycle.
3. *TLElaborationPhaseState* — the top level enters this state before starting each elaboration phase.
4. *TLElaborationCycleState* — the top level enters this state before starting each elaboration cycle.
5. *TLInputCycleState* — the top level uses this state to step the input cycle state machine.
6. *TLPreferencePhaseState* — the top level uses this state to step the preference phase state machine.
7. *TLWorkingMemoryPhaseState* — the top level uses this state to step the working memory phase state machine.

8. *TLOutputCycleState* — the top level uses this state to step the output cycle state machine.

9. *TLDecisionPhaseState* — the top level uses this state to step the quiescence phase state machine.

10. *TLFinishedState* — the state that the top level enters when the system is halted. There is no way to reach this state in the specification because Soar is a situated agent that behaves continuously, i.e., Soar never turns itself off. However, an implementation would have commands that terminate Soar's execution by moving the top level state machine into its *TFFinishedState*.

$$
\begin{aligned}
\mathit{TopLevelState} ::= \ & \mathit{TLInitialState} \\
\mid \ & \mathit{TLDecisionCycleState} \\
\mid \ & \mathit{TLElaborationPhaseState} \\
\mid \ & \mathit{TLElaborationCycleState} \\
\mid \ & \mathit{TLInputCycleState} \\
\mid \ & \mathit{TLPreferencePhaseState} \\
\mid \ & \mathit{TLWorkingMemoryPhaseState} \\
\mid \ & \mathit{TLOutputCycleState} \\
\mid \ & \mathit{TLDecisionPhaseState} \\
\mid \ & \mathit{TLFinishedState}
\end{aligned}
$$

The state schema for the top level, *TL* just holds a state counter of type *TopLevelState*.

```
__ TL ____________________________________
 | top_level_state : TopLevelState
 |_________________________________________
```

8.2 Soar's State Schema

The Soar state schema contains the state schemas for recognition memory, IO, and Decide. It also includes the state schema for the top level (*TL*). The *IdentifierTable* is defined to be the union of all of the component identifiers of the productions, temporary memory elements, components and traces.

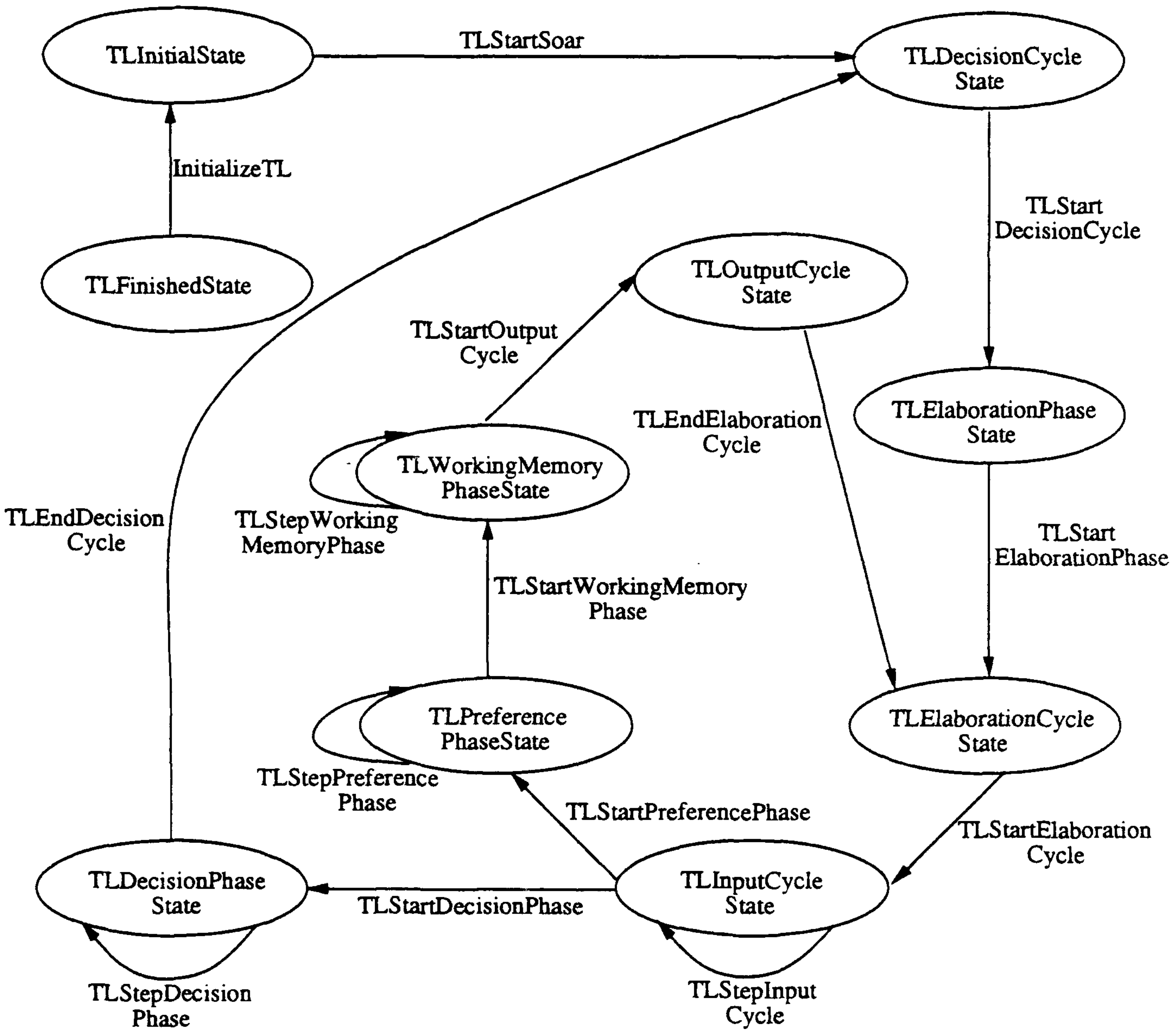

Figure 8.1: Top Level State Machine

Soar

RM
IO
$Decide$
TL
$IdentifierTable : \mathbb{P}\ Identifier$

$$
\begin{aligned}
& IdentifierTable = \\
& \quad (\bigcup(SpsComponents\,(\!|SPM|\!)\cup \\
& \qquad TMEsComponents\,(\!|TM|\!)\cup \\
& \qquad TracesComponents\,(\!|\operatorname{ran} TrP|\!))\cap \\
& \qquad\quad Identifier)
\end{aligned}
$$

The *InitSoar* schema specifies the valid initial configurations for Soar. It constrains Soar's initial configuration to be an initial configuration for recognition memory, IO and decide, and the top level state machine must start in the *TLInitialState* state.

InitSoar

$Soar$

$InitRM$

$InitIO$

$InitDecide$

$top\_level\_state = TLInitialState$

8.3 The Top Level State Machine

TLInitialize initializes the top level state machine. Soar never initializes itself: only the user interface will ever invoke this operation.

TLInitialize

$\Delta Soar$

$top\_level\_state = TLFinishedState$

$top\_level\_state' = TLInitialState$

TLReset resets the top level state machine to its initial configuration from any of its states. The top level state machine might have been stepping one or more of the hierarchy of sub-state machines, so it resets their states also.

TLReset

$\Delta Soar$

$top\_level\_state' = TLInitialState$

$PPhaseReset$

$InputCycleReset$

$OutputCycleReset$

$QPhaseReset$

$WMPhaseReset$

TLStartSoar moves Soar from its initial state into the *TLDecisionCycleState*. The *TLDecisionCycleState* is where each iteration of a decision cycle is started. When Soar is first started, this transition initializes decide's first goal with *DecideCreateFirstGoal*.

TLStartSoar

$\Delta Soar$

$top\_level\_state = TLInitialState$

$DecideCreateFirstGoal$

$top\_level\_state' = TLDecisionCycleState$

TLStartDecisionCycle moves from the *TLDecisionCycleState* into the *TLElaborationPhaseState* starting the first elaboration phase of the decision cycle.

TLStartDecisionCycle

$\Delta Soar$

$top\_level\_state = TLDecisionCycleState$

$top\_level\_state' = TLElaborationPhaseState$

The elaboration phase consists of many elaboration cycles; so the *TLStartElaborationPhase* transition moves the top level from the *TLElaborationPhaseState* into the *TLElaborationCycleState*.

TLStartElaborationPhase

$\Delta Soar$

$top\_level\_state = TLElaborationPhaseState$

$top\_level\_state' = TLElaborationCycleState$

Each elaboration cycle is an input cycle, then a preference phase, a working memory phase and finally an output cycle. *TLStartElaborationCycle* moves from the *TLElaborationCycleState* into the *TLInputCycleState* to start the elaboration cycle by starting the input cycle. It initializes the input cycle state machine for execution, so *SoarInputCycleState* can step the input cycle state machine.

$$\begin{array}{|l}
\textit{TLStartElaborationCycle} \\
\Delta \textit{Soar} \\
\hline
\textit{top\_level\_state} = \textit{TLElaborationCycleState} \\
\textit{InputCycleInitialize} \\
\textit{top\_level\_state}' = \textit{TLInputCycleState} \\
\hline
\end{array}$$

TLStepInputCycle moves the input cycle state machine one step.

$$\begin{array}{|l}
\textit{TLStepInputCycle} \\
\Delta \textit{Soar} \\
\hline
\textit{top\_level\_state} = \textit{TLInputCycleState} = \textit{top\_level\_state}' \\
\textit{InputCycleStep} \\
\hline
\end{array}$$

When the input cycle state machine has finished, if Soar is not at *Quiescence*, *TLStartPreferencePhase* moves to *TlPreferencePhaseState* to start the preference phase.

$$\begin{array}{|l}
\textit{TLStartPreferencePhase} \\
\Delta \textit{Soar} \\
\hline
\textit{top\_level\_state} = \textit{TLInputCycleState} \\
\neg \text{ pre } \textit{InputCycleStep} \\
\neg \textit{ Quiescence} \\
\textit{top\_level\_state}' = \textit{TLPreferencePhaseState} \\
\hline
\end{array}$$

If Soar is at *Quiescence* then *TLStartDecisionPhase* moves Soar to the *TLDecisionPhaseState* to start the decision, and initializes QPhase.

TLStartDecisionPhase

$\Delta Soar$

$top\_level\_state = TLInputCycleState$

$\neg$ pre $InputCycleStep$

$Quiescence$

$QPhaseInitialize$

$top\_level\_state' = TLDecisionPhaseState$

TLStepPreferencePhase steps the preference phase.

TLStepPreferencePhase

$\Delta Soar$

$top\_level\_state = TLPreferencePhaseState = top\_level\_state'$

$PPhaseStep$

When the preference phase has finished its work, *TLStartWorkingMemoryPhase* moves Soar from *TLPreferencePhaseState* into *TLWorkingMemoryPhaseState*.

TLStartWorkingMemoryPhase

$\Delta Soar$

$top\_level\_state = TLPreferencePhaseState$

$\neg$ pre $PPhaseStep$

$WMPhaseInitialize$

$top\_level\_state' = TLWorkingMemoryPhaseState$

TLStepWorkingMemoryPhase steps the working memory phase.

TLStepWorkingMemoryPhase

$\Delta Soar$

$top\_level\_state = TLWorkingMemoryPhaseState = top\_level\_state'$

$WMPhaseStep$

When the working memory phase has finished, *TLStartOutputCycle* initializes the output cycle and moves Soar to *TLOutputCycleState*.

$$\begin{array}{|l}
\textit{TLStartOutputCycle} \\
\hline
\Delta Soar \\
\hline
top\_level\_state = TLWorkingMemoryPhaseState \\
\neg \text{ pre } WMPhaseStep \\
top\_level\_state' = TLOutputCycleState \\
OutputCycleInitialize \\
\hline
\end{array}$$

TLStepOutputCycle steps the output cycle.

$$\begin{array}{|l}
\textit{TLStepOutputCycle} \\
\hline
\Delta Soar \\
\hline
top\_level\_state = TLOutputCycleState = top\_level\_state' \\
OutputCycleStep \\
\hline
\end{array}$$

When the output cycle has finished its work, *TLEndElaborationCycle* moves Soar back to the *ElaborationPhaseState* to start the next elaboration cycle of the elaboration phase.

$$\begin{array}{|l}
\textit{TLEndElaborationCycle} \\
\hline
\Delta Soar \\
\hline
top\_level\_state = TLOutputCycleState \\
\neg \text{ pre } OutputCycleStep \\
top\_level\_state' = TLElaborationCycleState \\
\hline
\end{array}$$

TlStepQPhase steps QPhase in the *DecisionPhaseState*.

$$\begin{array}{|l}
\textit{TLStepQPhase} \\
\hline
\Delta Soar \\
\hline
top\_level\_state = TLDecisionPhaseState = top\_level\_state' \\
QPhaseStep \\
\hline
\end{array}$$

When the quiescence phase has finished, *TLEndDecisionCycle* moves Soar back to the *DecisionCycleState* to start the next decision cycle of the decision phase.

$$\begin{array}{|l}
\hline
\textit{TLEndDecisionCycle} \\
\Delta \textit{Soar} \\
\hline
\textit{top\_level\_state} = \textit{TLDecisionPhaseState} \\
\neg\ \text{pre}\ \textit{QPhaseStep} \\
\textit{top\_level\_state}' = \textit{TLDecisionCycleState} \\
\hline
\end{array}$$

TLStep steps the entire execution of Soar by moving the top level state machine through one of its transitions.

$$\begin{aligned}
&\textit{TLStep} \mathrel{\widehat=} \textit{TLStartSoar} \lor \textit{TLStartDecisionCycle} \lor \\
&\qquad \textit{TLStartElaborationPhase} \lor \textit{TLStartElaborationCycle} \lor \\
&\qquad \textit{TLStepInputCycle} \lor \textit{TLStartPreferencePhase} \lor \\
&\qquad \textit{TLStartDecisionPhase} \lor \textit{TLStepPreferencePhase} \lor \\
&\qquad \textit{TLStartWorkingMemoryPhase} \lor \textit{TLStepWorkingMemoryPhase} \lor \\
&\qquad \textit{TLStartOutputCycle} \lor \textit{TLStepOutputCycle} \lor \\
&\qquad \textit{TLEndElaborationCycle} \lor \\
&\qquad \textit{TLStepQPhase} \lor \textit{TLEndDecisionCycle}
\end{aligned}$$

Appendix A

Numbers

We have specified a Soar that uses only constants and identifiers to construct its temporary memory structures. Ideally, the implementation would also use exactly the same data structures. Unfortunately, Soar users find it difficult to effectively process numbers without some implementation support. The long term solution is to provide problem spaces, complete with chunk libraries, that gracefully support numeric reasoning, and some work in this direction has already been completed. In the interim, we augment our implementation's base data structures with numeric constants and extend the test structure to let productions match against them with the basic arithmetic relations.

The implementation adds in:

$$\begin{array}{|l} \mathit{Number} : \mathbb{P}\ \mathit{Constant} \\ >, <, >=, <= : \mathit{SpecialSymbol} \\ \mathit{NumericRelationSymbols} : \mathbb{P}\ \mathit{SpecialSymbol} \\ \hline \mathit{NumericRelationSymbols} = \{>, <, >=, <=\} \end{array}$$

$$\mathit{Test} ::= \mathit{RelationalTest}\langle\langle \mathit{NumericRelationSymbols} \times (\mathit{Variable} \cup \mathit{Number})\rangle\rangle$$

$$\forall t : \operatorname{ran} \mathit{RelationalTest} \bullet \mathit{TestsComponents}(t) = \mathit{second}(\mathit{RelationalTest}^{\sim}(t))$$

$$\forall t : \operatorname{ran} \mathit{RelationalTest} \bullet \mathit{TestsPositiveComponents}(t) = \varnothing$$

We extended $\mathit{Matches}^{Test}_{Symbol}$ with some extra cases to handle numeric relations and same type tests must be extended to differentiate numbers from identifiers and other constants.

$$Matches_{Symbol}^{Test}\ \_ : \mathbb{P}(Binding \times Test \times Symbol)$$

$$
\begin{array}{l}
\forall\, b : Binding;\ t : Test;\ v : Symbol \bullet \\
\quad \mathrm{Matches}_{Symbol}^{Test}\ (b, t, v) \Leftrightarrow \\
\quad ((t \in \mathrm{ran}\ RelationalTest \wedge \\
\qquad (\exists\, nrsvn : \{RelationalTest^{\sim}(t)\} \bullet \\
\qquad\quad (\exists\, vn : \{((\mathrm{id}\ \Sigma) \oplus b)(second(nrsvn))\} \bullet \\
\qquad\qquad (\exists\, t : \{first(nrsvn)\} \bullet \\
\qquad\qquad\quad (t = > \Rightarrow v > vn) \wedge \\
\qquad\qquad\quad (t = < \Rightarrow v < vn) \wedge \\
\qquad\qquad\quad (t = >= \Rightarrow v >= vn) \wedge \\
\qquad\qquad\quad (t = <= \Rightarrow v <= vn))))) \vee \\
\qquad (t \in \mathrm{ran}\ SameTypeTest \wedge \\
\qquad\quad (\exists\, v2 : \{((\mathrm{id}\ Symbol) \oplus b)(SameTypeTest^{\sim}(t))\} \bullet \\
\qquad\qquad ((v \in Identifier \wedge v2 \in Identifier) \vee \\
\qquad\qquad\ (v \in Constant \setminus Number \wedge v2 \in Constant \setminus Number) \vee \\
\qquad\qquad\ (v \in Number \wedge v2 \in Number)))))
\end{array}
$$

Appendix B

An Implementation Discipline

Section B.1 details a technique to allow users to observe and modify the implementation of Soar. Section B defines an implementation discipline to govern implementations of the specification. The final section, Section B.3, calculates the worst case cost of actually using the state machine representation for Soar's control.

B.1 Observing Soar's Operation

As Soar is a research vehicle for cognitive modeling and artificial intelligence work ([New90]), we would like to have an implementation that we can easily monitor and modify. We require a specification and implementation discipline that helps us to support these goals. In previous implementations, if a researcher wanted to measure the frequency of a certain type of chunk creation, she would have to read large tracts of the source code and install her statistics directly in a copy of the code. As there was no other detailed enough description of the system, only the code was an effective index into the system's operation.

Coupling this specification with an implementation discipline would allow Soar researchers to more easily solve this problem. In [PMN91] we describe a technique called *Hooking* that allows the system to execute a function, called an *Observer*, before and after the execution of any sequential operation. The technique also allows the user to, at run-time, redefine any specified operation: as long as the new implementation meets this specification of the operation in this document. Researchers could then use this specification as a guide to place observers on hooks that monitor or change the implementation's operation.

B.2 An Implementation Discipline

This section defines an implementation discipline that maps parts of this specification into executable code. This discipline was designed to allow the easy observation and modification of Soar, but the new implementation of Soar, Soar6, is not constructed using this technique.

Ideally, we would like to semi-mechanically map specifications into an optimized implementation. However, the automation of this mapping is a difficult research problem in program synthesis, so we have settled for an informal manual mapping.

The Soar specification contains four types of objects that must be mapped into the implementation:

1. **state schemas** — defined using schemas
2. sequential operations — defined using schemas with Δ in their signature
3. data structures — defined using base types, free types and schemas
4. state machines — built of state schemas and sequential operations

This discipline maps each state schema into a structure of the same name in the implementation. These structures contain fields for the schema's components and may contain other fields for implementation-specific information.

Mapping Sequential Operations Each sequential operation is mapped to a function of the same name in the implementation. Some of the sequential operations take inputs using the Z "?" decoration convention ([Spi89] 133) and many non-deterministically select their arguments from sets using existential quantification. The observers are passed all arguments and top level existentially quantified variables to simplify their observation of the operation.

Associated with each state changing function are two sets of observer functions, called hook sets, and an optional implementation function. The observer functions in the first, or pre-hook set, are called before the operation function executes. The observer functions in the second, or post hook set, are called after the operation function executes. The observer functions may inspect any part of Soar's state, but may not change Soar's state. The observers allow users to instrument Soar's state changes. If an implementation function has been specified it is called instead of the default sequential operation function. This would allow researchers to easily change small pieces of the implementation.

Soar's specification currently has about 150 state changing operations and most users will never care to observe or modify any of them. At the time the Soar implementation is compiled, the user would be able to turn on only the hooks she requires. Any hooks not switched on would produce no code in the resulting implementation and so have no run-time cost. If an operation has been compiled to allow hooking, the hook set and implementation could be filled at run-time, allowing choice as to when as well as where to hook.

Mapping Data Structures The mapping to generate the data structures is much less mechanical. Implementations of data structures must be carefully selected for efficiency. Basic types would generally be represented using enumerations, structures or built-in types such as numbers. Schemas that define data types would mostly be represented using structures. Free types that define only constants would be represented as enumerated types. Free types that use injections to define types would generally be represented by structures using tags and union types. Mappings and sets would be represented in a variety of ways including pointers, lists, functions and hash tables.

Mapping State Machines The specification's state machines are constructed of state schemas and sequential operations. As above, each state schema would be represented by a structure and each transition by a function. However, many states of the machines have more than one transition leaving the state. The state machine's implementation must choose which of these transitions to apply. Every state that has more than one transition is mapped to a function of the same name that chooses which transition to apply.

Some transitions in the state machines check the pre-condition of the step operation of a sub-state machine. For example, *FinishOP* checks $\neg$ pre *StepS3*. The implementation could check these pre-conditions by checking the state counter of the sub-state machine. If the machine is in any state other than the final state, its pre-condition is satisfied.

Some transitions check the pre-condition of other operations: *StartS3* checks $\neg$ pre *S2*. The implementation is free to calculate this pre-condition in any way that does not allow a user of the system to observe performance different than that described by the specification. A fast way to check this is to have the state function (in this case, *S2State*'s function) record that the pre-conditions of *S2* no longer holds on *S2*'s last execution. For example, if *S2* is iterating over a set of inputs, when the set is empty it will mark that its pre-condition is no longer satisfied in a flag. When the *S2State* function is called to select the next transition, it will consult the pre-condition flag and select *StartS3*.

B.3 An Estimate of the Cost of Using State Machines

The use of state machines for control in an implementation would have a greater efficiency cost than using standard sequential code for control. This section uses an analytic and empirical analysis to demonstrate that the expense of state machine control is at worst a small percentage of Soar's total cost. First, we describe an implementation of the state machines in detail. Second, we identify the most expensive step of state machine control. Third, we prototype just this step and benchmark it. Fourth, we estimate the number of such steps in a large Soar run and compare our measured cost with execution times from Soar5 and optimistic speed-up factors for Soar6.

The implementation of each state machines has five parts:

1. a stepping function
2. a state name counter
3. an array that the stepping function uses to index into the state functions
4. a function for each state that selects which transition to apply
5. a function for each transition

For each transition of a state machine, first the stepping function is called. It indexes using the state name into the array of state functions and calls the function for the current state. This function calculates which of the possible transitions to execute, and applies one. Thus for each step of a state machine, Soar performs three function calls and one array access. A sequential control model would represent the stepping, the array access and the state function in sequential code.

At the deepest, the specification nests state machine control only three levels. The most expensive step is likely to be the state machine to calculate preference semantics, as it must execute once for every change to any temporary memory slot. The longest execution path through this state machine has 10 transitions between 11 states plus one initialization step. This is actually an overestimate of the number of control steps required. As long as the implementation produces the same results and has equivalent observability properties, the implementation would be free to optimize the average case by exiting this sequence of steps early. For every step of preference semantics, Soar would also step the two state machines that drive it: the working memory phase and the top level.

We constructed a prototype C implementation of this control flow. It contains all of the control flow that the top level state machine and the working memory phase state machine would require to step preference semantics in a cycle of 11 transitions. On a Decstation 5000[1] , the prototype ran 11 million transitions in 37 seconds. This is very likely an overestimate of an actual implementation's cost because the heavy calling of pointers to functions of this code pessimizes the use of the processor's instruction cache. When the control flow is augmented with calculation, the instruction pipeline should be able to execute some of the calculation instructions in parallel with the control flow.

The largest Soar 5.2 system that has been constructed to date runs approximately 235,000 RHS actions in 7,000 seconds on the same machine. Assuming

[1]Decstation 5000 is a trade mark of the Digital Equipment Corporation.

a roughly equal rate of calls to preference semantics, Soar 5.2 requires approximately 33.6 calls to preference semantics a second, or 403.2 total control steps per second. Suppose also that we are underestimating the number of total control steps by a factor of two. This means that for every time the system takes one step inside preference semantics, it takes two steps in running the other modules of the system. Then it would requires 806.4 total control steps per second of computation. Suppose very optimistically that Soar6 is 10 times faster than Soar 5.2. Soar6 would then require 8064 total state machine control steps per second. At 37 seconds per 11 million steps in Soar6, the system would spend about 2.7% of its time running the state machines.

This implementation discipline directly maps the state machines into the implementation for two reasons: they allow the system to breakpoint execution at a fine grain, and they simplify the mapping from the specification to the code. The ability to potentially breakpoint at any state in the system allows users to precisely select debugging breakpoints, stop the system, examine its current state and continue. The breakpointing will also allow the user to step the system in small increments providing a very detailed observation of the system that would facilitate learning and debugging. The small cost of at worst a few percent of total runtime seems a small cost to pay for the benefits of breakpointing.

Appendix C

Finer Grained Hooks

The granularity of the sequential operations in the specification of Soar is too coarse to allow some types of observation of the system, using the state machine implementation discipline. For example, if a researcher wanted to replace the implementation of working memory with a new data structure, she would have to replace the implementation of every operation that changed the contents of working memory. This requires re-implementing at least 13 operations.

We would like the specification to be structured to localize access to data structures to a few operations: like a program constructed using the data object module paradigm. If we were to construct sequential operations such as *AddWME*, which added an argument working memory element to working memory, we could still not easily apply them because Z's sequential system model is too weak to concisely describe the common operation of iterating *AddWME* across a set of elements.

Instead of not specifying the fine grained operations, and hence not allowing their hooking, we work around this problem by requiring the implementation to use a special collection of fine grained accessors. All of the modifications to *TM*, *WM*, *PM*, *SPM*, and *IM* or *GM* will call the operations specified in this chapter. Where these sets are modified by set unions or differences, an implementation is required to repeatedly apply the individual addition and deletion operations.

C.1 TM Hooks

Temporary memory is hooked at the working memory level, the preference memory level and the temporary memory level.

C.1.1 WME Operations

AddWME adds its argument element to working memory. The element can already be in working memory, in which case it is not re-added.

AddWME

ΔRM
$w? : WME$

$WM' = WM \cup \{w?\}$

RemoveWME removes its argument element from working memory. The element must already exist in memory, or an implementation specific error condition is signalled.

RemoveWME

ΔRM
$w? : WME$

$w? \in WM$
$WM' = WM \setminus \{w?\}$

C.1.2 Preference Operations

The operation *AddPreference* adds its preference to preference memory. The preference may already exists in memory, in which case preference memory is not changed.

AddPreference

ΔRM
$p? : Preference$

$PM' = PM \cup \{p?\}$

The *RemovePreference* operation removes its preference from preference memory. The preference must be in memory or an implementation specific error condition is signalled.

RemovePreference

ΔRM
$p? : Preference$

$p? \in PM$
$PM' = PM \setminus \{p?\}$

C.1.3 TME Operations

AddTME adds its element to temporary memory. If the element is already in memory, temporary memory is not changed.

AddTME

ΔRM

$t? : TME$

$TM' = TM \cup \{t?\}$

RemoveTME removes its element from temporary memory. The element must be in temporary memory or an implementation specific error condition is signalled.

RemoveTME

ΔRM

$t? : TME$

$t? \in TM$

$TM' = TM \setminus \{t?\}$

The implementations of *AddWME*, *RemoveWME*, *AddPreference*, *RemovePreference*, *AddTME* and *RemoveTME* respect the invariant that temporary memory is constructed from working and preference memory. For example, whenever an operation calls *AddWME* the equivalent *AddTME* must also be called. Whenever an operation calls *RemoveTME* the equivalent *RemoveWME* or *RemovePreference* must also be called.

C.2 SPM Hooks

The Soar production memory, *SP*, has productions added and removed by name. The *AddSP* operation will only add in a production if there is not already one in memory of that name.

AddSP

ΔRM

$p? : SP$

$\neg\, (\exists\, p : SPM \bullet p.name = p?.name)$

$SPM' = SPM \cup \{p?\}$

The *RemoveSP* operation will remove a production given its name.

RemoveSP

$$\begin{array}{l} \Delta RM \\ pname : Symbol \\ \hline \exists\, p : SPM \mid p.name = pname \bullet \\ \quad SPM' = SPM \setminus \{p\} \end{array}$$

C.3 GM and IM Hooks

The goal and impasse memory changes can be hooked only for context impasses, for non-context impasses or for any impasse using the six operations specified in this section.

C.3.1 Context Impasses

There is no need for an operation that hooks the addition of context impasses, because the specification contains only two operations that install them: *DecideCreateFirstGoal* and *ImpasserCreateImpasse*.

Although there is only one specified operation that removes an impasse, *ImpasserRemoveImpase*, a user interface to Soar requires an operation to pop goals from the goal stack. *RemoveContextImpasse* is specified for this purpose. It removes a context impasse from goal memory, all of its descendent goals and all of their *ImpasseWMEs*. The implementation should behave as if a sequence of *RemoveContextImpasse* operations occur, one for each of the goals that would be removed to match this specification, from newest to oldest.

RemoveContextImpasse

$$\begin{array}{l} \Delta Decide \\ i\_or\_g? : Identifier \\ \hline i\_or\_g? \in GM \\ GM' = (GM \setminus \{i\_or\_g?\}) \setminus GoalDescendants(i\_or\_g?, GM, WM) \\ WM' = (WM \setminus ImpassesWMEs(i\_or\_g?, GM, WM)) \\ \quad \setminus (\bigcup \{g : GoalDescendants(i\_or\_g?, GM, WM) \bullet \\ \qquad ImpassesWMEs(g, GM, WM)\}) \end{array}$$

C.3.2 Non-Context Impasses

Non-context impasses are only generated in *ImpasserCreateImpasse*, and only explicitly removed by *ImpasserRemoveImpasse*. However, some time after an impasse becomes disconnected from the context stack, it will be removed from impasse memory and it will have its working memory elements removed from memory. For convenient hooking of this removal, *RemoveNonContextImpasse* is defined to remove the impasse from impasse memory and remove its working memory elements.

```
── RemoveNonContextImpasse ──────────────────────
  $\Delta Decide$
  $i\_or\_g? : Identifier$
├──────────────
  $i\_or\_g? \in IM$
  $IM' = IM \setminus \{i\_or\_g?\}$
  $WM' = WM \setminus ImpassesWMEs(i\_or\_g?, IM, WM)$
└────────────────────────────────────────
```

Appendix D

The Reorderer

The reorderer is a heuristic greedy search algorithm. It searches the space of possible orderings of a production's condition elements for one that will produce a low average match cost. The reorderer does not improve the match cost of cheap chunks very much, but it is important to reduce the efficiency loss of expensive chunks.

This specification is the result of an effort to carefully understand and document the branching factor heuristic of the greedy search. The previous implementation[Sca86] lumped the conditions into five equivalence classes without carefully understanding the nature of the heuristic or the fine grained test structure of the conditions. Consequently, the calculation of the heuristic was mixed into the control structure of the search. It could not be tailored to fully take advantage of architecture and user supplied information. Worse still it threw up its hands on complicated conjunctive or simple disjunctive tests and assigned them to an expensive equivalence class.

The reorderer has three parts: a classifier, a branching factor heuristic and the greedy search itself. The classifier groups working memory element tests into classes based upon the constants and variables tested in their components and the set of bound variables and the set of variables constrained to bind to impasse identifiers. The classification is used to estimate the average match cost of the condition, called the branching factor [Tam91]. The reorderer performs a greedy search for an ordering of conditions that minimizes the branching factor of the entire ordering; with ties broken by two steps of look-ahead.

D.1 Classifying WMETests

Working memory element tests are classified bottom up: the constants of the tests are classified, then the test is classified, and finally the working memory element test is classified.

The constants and tests of the conditions are classified into seven classes:

- C— a test that tests for equality with a constant
- B— a test that tests for equality with a bound variable
- U— a test that tests for equality with an unbound variable or one that tests only for inequality with a constant or bound variable
- GIU— a test that tests for equality with an unbound goal or impasse variable
- GIB— a test that tests for equality with a bound goal or impasse variable
- YES— a test for an acceptable preference and
- NO— a test for not being an acceptable preference.

The set *TestClass* is constructed from Σ to hold the set of the possible classes.

$$\begin{array}{l} \mathrm{C, B, U, GIU, GIB, YES, NO} : \Sigma \\ \mathit{TestClass} : \mathbb{P}\,\Sigma \\ \hline \mathit{TestClass} = \{\mathrm{C, B, U, GIU, GIB, YES, NO}\} \end{array}$$

ClassifySigma takes a constant or variable from a test, the set of bound variables and the set of variables bound to impasses and maps them into one of the seven classifications.

$$\begin{array}{l} \mathit{ClassifySigma} : \Sigma \times \mathbb{P}\,\mathit{Variable} \times \mathbb{P}\,\mathit{Variable} \rightarrow \mathit{TestClass} \\ \hline \forall\, vc : \Sigma;\ \mathit{bound}, \mathit{impasses} : \mathbb{P}\,\mathit{Variable} \bullet \\ \quad ((vc \in \mathit{Constant} \Rightarrow \mathit{ClassifySigma}(vc, \mathit{bound}, \mathit{impasses}) = \mathrm{C}) \wedge \\ \quad\; (vc \in \mathit{Variable} \Rightarrow \\ \quad\quad (vc \in \mathit{impasses} \Rightarrow \\ \quad\quad\quad (vc \in \mathit{bound} \Rightarrow \\ \quad\quad\quad\quad \mathit{ClassifySigma}(vc, \mathit{bound}, \mathit{impasses}) = \mathrm{GIB}) \wedge \\ \quad\quad\quad (vc \notin \mathit{bound} \Rightarrow \\ \quad\quad\quad\quad \mathit{ClassifySigma}(vc, \mathit{bound}, \mathit{impasses}) = \mathrm{GIU})) \wedge \\ \quad\quad (vc \notin \mathit{impasses} \Rightarrow \\ \quad\quad\quad (vc \in \mathit{bound} \Rightarrow \\ \quad\quad\quad\quad \mathit{ClassifySigma}(vc, \mathit{bound}, \mathit{impasses}) = \mathrm{B}) \wedge \\ \quad\quad\quad (vc \notin \mathit{bound} \Rightarrow \\ \quad\quad\quad\quad \mathit{ClassifySigma}(vc, \mathit{bound}, \mathit{impasses}) = \mathrm{U})))) \end{array}$$

The *ClassifyTest* function takes a test, the set of bound variables and the set of variables bound to impasses and classifies the test. Blank tests are classified as if they were unbound variables because, like an unbound variable, they can match anything. Yes tests and No tests are classified into YES and NO classes. Equality tests are classified by the variable or constant that they match. Same-type tests and not tests can match against an arbitrary number of values, so they are also classified as unbound variables. Disjunctions can only match against one of a few constants, so they are classified as constants.

Conjunctions are more complicated to classify. If the conjunction contains a disjunction it can only match the constants in the disjunction, so it is classified as a constant. If it contains an equality it is assigned the classification of the equality. If it contains a not test or a same-type test, it is treated as an unbound variable. These rules are not mutually exclusive; they are intended to be applied sequentially to produce the most match restrictive classification.

$$
\begin{array}{l}
\mathit{ClassifyTest} : \mathit{Test} \times \mathbb{P}\ \mathit{Variable} \times \mathbb{P}\ \mathit{Variable} \rightarrow \mathit{TestClass} \\
\hline
\forall\, t : \mathit{Test};\ \mathit{bound}, \mathit{impasses} : \mathbb{P}\ \mathit{Variable} \bullet \\
\quad ((t = \mathit{BlankTest} \Rightarrow \mathit{ClassifyTest}(t, \mathit{bound}, \mathit{impasses}) = \mathrm{U}) \wedge \\
\quad (t = \mathit{YesTest} \Rightarrow \mathit{ClassifyTest}(t, \mathit{bound}, \mathit{impasses}) = \mathrm{YES}) \wedge \\
\quad (t = \mathit{NoTest} \Rightarrow \mathit{ClassifyTest}(t, \mathit{bound}, \mathit{impasses}) = \mathrm{NO}) \wedge \\
\quad (t \in \operatorname{ran} \mathit{EqualityTest} \Rightarrow \\
\qquad \mathit{ClassifyTest}(t, \mathit{bound}, \mathit{impasses}) = \\
\qquad\qquad \mathit{ClassifySigma}(\mathit{EqualityTest}^{\sim}(t), \mathit{bound}, \mathit{impasses})) \wedge \\
\quad (t \in \operatorname{ran} \mathit{NotTest} \Rightarrow \mathit{ClassifyTest}(t, \mathit{bound}, \mathit{impasses}) = \mathrm{U}) \wedge \\
\quad (t \in \operatorname{ran} \mathit{SameTypeTest} \Rightarrow \mathit{ClassifyTest}(t, \mathit{bound}, \mathit{impasses}) = \mathrm{U}) \wedge \\
\quad (t \in \operatorname{ran} \mathit{DisjunctiveTest} \Rightarrow \mathit{ClassifyTest}(t, \mathit{bound}, \mathit{impasses}) = \mathrm{C}) \wedge \\
\quad (t \in \operatorname{ran} \mathit{ConjunctiveTest} \Rightarrow \\
\qquad ((\exists\, c : \mathit{ConjunctiveTest}^{\sim}(t) \bullet c \in \operatorname{ran} \mathit{DisjunctiveTest}) \Rightarrow \\
\qquad \mathit{ClassifyTest}(t, \mathit{bound}, \mathit{impasses}) = \mathrm{C}) \wedge \\
\qquad ((\exists\, c : \mathit{ConjunctiveTest}^{\sim}(t) \bullet c \in \operatorname{ran} \mathit{EqualityTest}) \Rightarrow \\
\qquad\quad (\exists\, c : \mathit{ConjunctiveTest}^{\sim}(t) \cap \operatorname{ran} \mathit{EqualityTest} \bullet \\
\qquad\qquad \mathit{ClassifyTest}(t, \mathit{bound}, \mathit{impasses}) = \\
\qquad\qquad\qquad \mathit{ClassifySigma}(\mathit{EqualityTest}^{\sim}(c), \mathit{bound}, \mathit{impasses}))) \wedge \\
\qquad ((\exists\, c : \mathit{ConjunctiveTest}^{\sim}(t) \bullet c \in \operatorname{ran} \mathit{NotTest}) \\
\qquad\quad \Rightarrow \mathit{ClassifyTest}(t, \mathit{bound}, \mathit{impasses}) = \mathrm{U}) \wedge \\
\qquad ((\exists\, c : \mathit{ConjunctiveTest}^{\sim}(t) \bullet c \in \operatorname{ran} \mathit{SameTypeTest}) \Rightarrow \\
\qquad\quad \mathit{ClassifyTest}(t, \mathit{bound}, \mathit{impasses}) = \mathrm{U})))
\end{array}
$$

The *WMETestClass* schema holds classifications for an entire *WMEtest*. The WME test class uses Σ as the value of the attribute and value components so that the classes can be used to map attribute and value specific classifications to branching factors. The system and the user can then enter default information about the branching factors of conditions in a uniform manner.

$$
\begin{array}{|l}
\hline
\textit{WMETestClass} \\
\textit{id} : \textit{TestClass} \\
\textit{attribute} : \Sigma \\
\textit{value} : \Sigma \\
\textit{context\_acceptable\_preference} : \{\mathrm{YES}, \mathrm{NO}\} \\
\hline
\end{array}
$$

The description of the derivation of the branching factor table requires the instantiation of 90 WME test classes. For brevity, the *WC* constructor is defined to instantiate the WME test classes.

$$
\begin{array}{|l}
\textit{WC} : \textit{TestClass} \times \textit{TestClass} \times \textit{TestClass} \times \textit{TestClass} \rightarrow \textit{WMETestClass} \\
\hline
\forall\, \textit{id} : \textit{TestClass};\ \textit{attribute} : \textit{TestClass};\ \textit{value} : \textit{TestClass}; \\
\quad \textit{context\_acceptable\_preference} : \{\mathrm{YES}, \mathrm{NO}\} \bullet \\
\quad \textit{WC}(\textit{id}, \textit{attribute}, \textit{value}, \textit{context\_acceptable\_preference}) = \\
\qquad (\mu\, \textit{id} : \{\textit{id}\};\ \textit{attribute} : \{\textit{attribute}\};\ \textit{value} : \{\textit{value}\}; \\
\qquad\quad \textit{context\_acceptable\_preference} : \{\textit{context\_acceptable\_preference}\} \bullet \\
\qquad \theta\, \textit{WMETestClass})
\end{array}
$$

ClassifyWMETest classifies a WME test given the set of bound variables and the set of variables bound to impasses. For the attribute and values the goal and impasse variables are not passed to *ClassifyTest* so that classify test will classify goal variables in these slots as normal variables.

$$
\begin{array}{|l}
\textit{ClassifyWMETest} : \textit{WMETest} \times \mathbb{P}\ \textit{Variable} \times \mathbb{P}\ \textit{Variable} \rightarrow \textit{WMETestClass} \\
\hline
\forall\, \textit{wt} : \textit{WMETest};\ \textit{bound}, \textit{impasses} : \mathbb{P}\ \textit{Variable} \bullet \\
\quad \textit{ClassifyWMETest}(\textit{wt}, \textit{bound}, \textit{impasses}) = \\
\qquad \textit{WC}(\textit{ClassifyTest}(\textit{wt.id}, \textit{bound}, \textit{impasses}), \\
\qquad\qquad \textit{ClassifyTest}(\textit{wt.attribute}, \textit{bound}, \varnothing), \\
\qquad\qquad \textit{ClassifyTest}(\textit{wt.value}, \textit{bound}, \varnothing), \\
\qquad\qquad \textit{ClassifyTest}(\textit{wt.context\_acceptable\_preference}, \varnothing, \varnothing))
\end{array}
$$

D.2 Deriving the Branching Factor Heuristic

The branching factor is a measure of the average cost to match a single condition, given the set of bound variables and the set of variables bound to impasses. The principled derivation of this heuristic is complicated. There are 90 (5 * 3 * 3 *

2) equivalence classes of WME test; as the identifier slot of a WME test can be classified five ways, the attribute and value each three ways and the preference two ways.

The analysis of these ninety cases into branching factors is derived from five measures of the average branching of the graph of working memory.

1. G — the average number of goal and impasses in the system.
2. O — the average number of objects in the system.
3. A — the average number of attributes per object in the system.
4. V — the average number of values for any attribute in the system.
5. AP — the average number of context acceptable preferences per values.

$$G, O, A, V, AP : \mathbb{Z}$$

$$G > 0 \wedge O > 0 \wedge O > 0 \wedge V > 0 \wedge AP > 0$$

The branching factor for each WME test is calculated by multiplying together the average measure for each slot that the condition leaves unconstrained.

The following table derives the average case match estimate for all 90 cases. This table is provided only to allow discussion of the correctness of this measure. The implementation will use a closed form of this function that is specified next.

1. WC(C,C,C,YES) = 1
2. WC(C,C,C,NO) = 1
3. WC(C,C,B,YES) = 1
4. WC(C,C,B,NO) = 1
5. WC(C,C,U,YES) = V * AP
6. WC(C,C,U,NO) = V
7. WC(C,B,C,YES) = 1
8. WC(C,B,C,NO) = 1
9. WC(C,B,B,YES) = 1
10. WC(C,B,B,NO) = 1
11. WC(C,B,U,YES) = V * AP
12. WC(C,B,U,NO) = V

13. WC(C,U,C,YES) = A

14. WC(C,U,C,NO) = A

15. WC(C,U,B,YES) = A

16. WC(C,U,B,NO) = A

17. WC(C,U,U,YES) = A * V * AP

18. WC(C,U,U,NO) = A * V

19. WC(B,C,C,YES) = 1

20. WC(B,C,C,NO) = 1

21. WC(B,C,B,YES) = 1

22. WC(B,C,B,NO) = 1

23. WC(B,C,U,YES) = V * AP

24. WC(B,C,U,NO) = V

25. WC(B,B,C,YES) = 1

26. WC(B,B,C,NO) = 1

27 WC(B,B,B,YES) = 1

28. WC(B,B,B,NO) = 1

29. WC(B,B,U,YES) = V * AP

30. WC(B,B,U,NO) = V

31 WC(B,U,C,YES) = A

32. WC(B,U,C,NO) = A

33. WC(B,U,B,YES) = A

34. WC(B,U,B,NO) = A

35. WC(B,U,U,YES) = A * V * AP

36. WC(B,U,U,NO) = A * V

37. WC(U,C,C,YES) = O

38. WC(U,C,C,NO) = O

39. WC(U,C,B,YES) = O

40. WC(U,C,B,NO) = O

41. WC(U,C,U,YES) = O * V * AP

42. WC(U,C,U,NO) = O * V

43. WC(U,B,C,YES) = O

44. WC(U,B,C,NO) = O

45. WC(U,B,B,YES) = O

46. WC(U,B,B,NO) = O

47. WC(U,B,U,YES) = O * V * AP

48. WC(U,B,U,NO) = O * V

49. WC(U,U,C,YES) = O * A

50. WC(U,U,C,NO) = O * A

51. WC(U,U,B,YES) = O * A

52. WC(U,U,B,NO) = O * A

53. WC(U,U,U,YES) = O * A * V * AP

54. WC(U,U,U,NO) = O * A * V

55. WC(GIU,C,C,YES) = G

56. WC(GIU,C,C,NO) = G

57. WC(GIU,C,B,YES) = G

58. WC(GIU,C,B,NO) = G

59. WC(GIU,C,U,YES) = G * V * AP

60. WC(GIU,C,U,NO) = G * V

61. WC(GIU,B,C,YES) = G

62. WC(GIU,B,C,NO) = G

63. WC(GIU,B,B,YES) = G

64. WC(GIU,B,B,NO) = G

65. WC(GIU,B,U,YES) = G * V * AP

66. WC(GIU,B,U,NO) = G

67. WC(GIU,U C,YES) = G * A

68. WC(GIU,U,C,NO) = G * A

69. WC(GIU,U,B,YES) = G * A

70. WC(GIU,U,B,NO) = G * A

71. WC(GIU,U,U,YES) = G * A * V * AP

72. WC(GIU,U,U,NO) = G * A * V

73. WC(GIB,C,C,YES) = 1

74. WC(GIB,C,C,NO) = 1

75. WC(GIB,C,B,YES) = 1

76. WC(GIB,C,B,NO) = 1

77. WC(GIB,C,U,YES) = V * AP

78. WC(GIB,C,U,NO) = V

79. WC(GIB,B,C,YES) = 1

80. WC(GIB,B,C,NO) = 1

81. WC(GIB,B,B,YES) = 1

82. WC(GIB,B,B,NO) = 1

83. WC(GIB,B,U,YES) = V * AP

84. WC(GIB,B,U,NO) = V

85. WC(GIB,U,C,YES) = A

86. WC(GIB,U,C,NO) = A

87. WC(GIB,U,B,YES) = A

88. WC(GIB,U,B,NO) = A

89. WC(GIB,U,U,YES) = A * V * AP

90. WC(GIB,U,U,NO) = A * V

This formulation is overly-general because each production's conditions must all be connected to an impasse. So all of the cases of the form $WC(\mathrm{U}, \ldots)$ will never be used. The current system assumes that any conditions of the form: $WC(\mathrm{B}, \mathrm{C}, \mathrm{U}, \mathrm{NO})$ are uni-attributes, e.g., have a branching factor of 1, unless told otherwise with a multi-attributes declaration.

The branching factor table of 90 cases is too large for easy use. A closed form for this table, named *ABF*, can be derived from the observation that the average branching factor of a WME test class is the average branching factor of its identifier (*ABFI*) multiplied by the average branching factor of its attribute (*ABFA*) and the average branching factor of its value acceptable preference combination (*ABFVAP*).

$$
\begin{array}{l}
ABF, ABFI, ABFA, ABFVAP : WMETestClass \rightarrow \mathbb{Z} \\
\hline
\forall\, wtc : WMETestClass \bullet \\
\quad ABF(wtc) = ABFI(wtc) * ABFA(wtc) * ABFVAP(wtc) \\
\\
\forall\, wtc : WMETestClass \bullet \\
\quad ((wtc.id \in \{\mathrm{C}, \mathrm{B}, \mathrm{GIB}\} \Rightarrow ABFI(wtc) = 1) \wedge \\
\quad\; (wtc.id = \mathrm{GIU} \Rightarrow ABFI(wtc) = G) \wedge \\
\quad\; (wtc.id = \mathrm{U} \Rightarrow ABFI(wtc) = O)) \\
\\
\forall\, wtc : WMETestClass \bullet \\
\quad ((wtc.attribute = \mathrm{U} \Rightarrow ABFA(wtc) = A) \wedge \\
\quad\; (wtc.attribute \neq \mathrm{U} \Rightarrow ABFA(wtc) = 1)) \\
\\
\forall\, wtc : WMETestClass \bullet \\
\quad ((wtc.value \neq \mathrm{U} \Rightarrow ABFVAP(wtc) = 1) \wedge \\
\quad\; (wtc.value = \mathrm{U} \wedge wtc.context\_acceptable\_preference = \mathrm{YES} \\
\quad\quad \Rightarrow ABFVAP(wtc) = V * AP) \wedge \\
\quad\; (wtc.value = \mathrm{U} \wedge wtc.context\_acceptable\_preference = \mathrm{NO} \\
\quad\quad \Rightarrow ABFVAP(wtc) = V))
\end{array}
$$

Soar users may care to provide information about branching factors of specific condition classes, and the system should have default knowledge about conditions that match context stack attributes. The user only cares to provide an average branching factor for the multi-attribute cases:
$WC(\mathrm{B}, \mathrm{C}, \mathrm{U}, \mathrm{NO})$ and $WC(\mathrm{B}, \mathrm{C}, \mathrm{U}, \mathrm{YES})$.

Using this classification scheme, the system can take advantage of some knowledge of the average case structure of the context stack:

- $WC(\mathrm{GIU}, \text{'OBJECT'}, \text{'NIL'}, \mathrm{NO}) = 1$ and $WC(\mathrm{GIB}, \text{'OBJECT'}, \text{'NIL'}, \mathrm{NO}) = 1$ — there is only one goal with no supergoal.

- $WC(\mathrm{GIU}, \text{'OBJECT'}, \mathrm{U}, \mathrm{NO}) = 1$ and $WC(\mathrm{GIB}, \text{'OBJECT'}, \mathrm{U}, \mathrm{NO}) = 1$ — there is only one supergoal for each goal.

- $WC(\text{GIU}, \text{`PROBLEM-SPACE'}, \text{U}, \text{NO}) = G$,
  $WC(\text{GIU}, \text{`STATE'}, \text{U}, \text{NO}) = G$ and
  $WC(\text{GIU}, \text{`OPERATOR'}, \text{U}, \text{NO}) = G$ — on average every goal has a problem space, state and operator.
- $WC(\text{GIB}, \text{`PROBLEM-SPACE'}, \text{U}, \text{NO}) = 1$,
  $WC(\text{GIB}, \text{`STATE'}, \text{U}, \text{NO}) = 1$ and
  $WC(\text{GIB}, \text{`OPERATOR'}, \text{U}, \text{NO}) = G$ — each specific goal has on average one problem-space, state or operator.
- $WC(\text{GIB}, \text{`ITEM'}, \text{U}, \text{NO}) > 1$ — on average the items augmentation of a goal holds more than one thing.

The user branching factor function, *UBF*, is provided to capture the user's knowledge of condition branching factors. A user interface, like Soar5's multi-attributes ([LCAS90] 178), should be provided to allow the user to extend *UBF*.

$$UBF : WMETestClass \rightarrow \mathbb{Z}$$

The system stores its knowledge of branching factors in the default branching factor function, *DBF*. *DBF* assumes that the $WC(\text{B}, \text{C}, \text{U}, \text{NO})$ case is a uni-attribute.

$$DBF : WMETestClass \rightarrow \mathbb{Z}$$

$$\begin{aligned}
DBF = \{&(WC(\text{GIU}, \text{`OBJECT'}, \text{`NIL'}, \text{NO}), 1),\\
&(WC(\text{GIB}, \text{`OBJECT'}, \text{`NIL'}, \text{NO}), 1),\\
&(WC(\text{GIU}, \text{`OBJECT'}, \text{U}, \text{NO}), 1),\\
&(WC(\text{GIB}, \text{`OBJECT'}, \text{U}, \text{NO}), 1),\\
&(WC(\text{GIU}, \text{`PROBLEM-SPACE'}, \text{U}, \text{NO}), G),\\
&(WC(\text{GIU}, \text{`STATE'}, \text{U}, \text{NO}), G),\\
&(WC(\text{GIU}, \text{`OPERATOR'}, \text{U}, \text{NO}), G),\\
&(WC(\text{GIB}, \text{`PROBLEM-SPACE'}, \text{U}, \text{NO}), 1),\\
&(WC(\text{GIB}, \text{`STATE'}, \text{U}, \text{NO}), 1),\\
&(WC(\text{GIB}, \text{`OPERATOR'}, \text{U}, \text{NO}), 1),\\
&(WC(\text{GIB}, \text{`ITEM'}, \text{U}, \text{NO}), 2),\\
&(WC(\text{B}, \text{C}, \text{U}, \text{NO}), 1)\}
\end{aligned}$$

BF takes two WME tests for arguments. The first is the WME test generated by the classifier. The second is the classifier with one of the possible constants of the attribute and/or value tests installed instead. *BF* checks if the system or the user has any constant specific branching information. If so it is used, otherwise the system uses the average branching factor.

$$
\begin{array}{|l}
BF : WMETestClass \times WMETestClass \rightarrow \mathbb{Z} \\
\hline
\forall\, prototype, instantiated : WMETestClass \bullet \\
\quad (instantiated \in \mathrm{dom}(DBF \oplus UBF) \Rightarrow \\
\qquad BF(prototype, instantiated) = (DBF \oplus UBF)(instantiated)) \land \\
\quad (instantiated \notin \mathrm{dom}(DBF \oplus UBF) \Rightarrow \\
\qquad BF(prototype, instantiated) = ABF(prototype))
\end{array}
$$

The reorderer must check the constant specific branching factor information for any positive constants that appear in attribute or value tests.

$$
\begin{array}{|l}
ContainsPositiveConstants\_ : \mathbb{P}\ Test \\
\hline
\forall\, t : Test \bullet \\
\quad ContainsPositiveConstants(t) \Leftrightarrow \\
\qquad (t \in \mathrm{ran}\ DisjunctiveTest \lor \\
\qquad\ t \in \mathrm{ran}\ EqualityTest \land EqualityTest^{\sim}(t) \in Constant \lor \\
\qquad t \in \mathrm{ran}\ ConjunctiveTest \land \\
\qquad\quad (\exists\, c : ConjunctiveTest^{\sim}(t) \bullet c \in \mathrm{ran}\ DisjunctiveTest \lor \\
\qquad\qquad (c \in \mathrm{ran}\ EqualityTest \land EqualityTest^{\sim}(c) \in Constant)))
\end{array}
$$

The Soar5 reorderer would not correctly classify many conditions that involved disjunctions or conjunctions in the attributes. *SBF* carefully picks the maximum branching factor for the set of constants of the attribute or value tests of the WME test.

$$
\begin{array}{|l}
SBF : WMETestClass \times WMETest \rightarrowtail \mathbb{Z} \\
\hline
\forall\, wtc : WMETestClass;\ wt : WMETest \bullet \\
\quad ((ContainsPositiveConstants(wt.attribute) \wedge \\
\quad\quad ContainsPositiveConstants(wt.value) \Rightarrow \\
\quad\quad SBF(wtc, wt) = \\
\quad\quad\quad\quad max\{a : TestsPositiveComponents(wt.attribute) \cap Constant; \\
\quad\quad\quad\quad\quad v : TestsPositiveComponents(wt.value) \cap Constant \bullet \\
\quad\quad\quad\quad\quad BF(wtc,\ WC(wtc.id, a, v, wtc.context\_acceptable\_preference))\}) \wedge \\
\quad (ContainsPositiveConstants(wt.attribute) \wedge \\
\quad\quad \neg\ ContainsPositiveConstants(wt.value) \Rightarrow \\
\quad\quad SBF(wtc, wt) = \\
\quad\quad\quad\quad max\{a : TestsPositiveComponents(wt.attribute) \cap Constant \bullet \\
\quad\quad\quad\quad\quad BF(wtc,\ WC(wtc.id, a, wtc.value, wtc.context\_acceptable\_preference))\}) \wedge \\
\quad (\neg\ ContainsPositiveConstants(wt.attribute) \wedge \\
\quad\quad \neg\ ContainsPositiveConstants(wt.value) \Rightarrow \\
\quad\quad SBF(wtc, wt) = ABF(wtc) = BF(wtc, wtc)))
\end{array}
$$

SBF will allow the reorderer to correctly order conditions, like this one from the default productions, containing the common idiom of binding to a disjunction.

```
(<s> ^{ << required-success success partial-success
                ... >> <svalue> } <eb> )
```

In this case, the reorderer can tell that this disjunction is a uni-attribute, not a multi-attribute.

The concept of a WME test's branching factor can be elevated to conditions. The branching factor of a positive condition is the branching factor of its WME test. Negative conditions can not be added until all of the variables that they share with the positive conditions have been bound. When a negative condition is added, its branching factor is one because it only filters the set of matches. For convenience, if the negative condition does not have all of its variables bound it is assigned an arbitrarily huge branching factor.

$$
\begin{array}{l}
\mathit{ConditionsBranchingFactor} : \\
\quad \mathit{Condition} \times \mathbb{P}\ \mathit{Variable} \times \mathbb{P}\ \mathit{Variable} \times \mathbb{P}\ \mathit{Variable} \rightarrow \mathbb{Z} \\
\hline
\forall\, c : \mathit{Condition};\ \mathit{bound}, \mathit{positive}, \mathit{impasses} : \mathbb{P}\ \mathit{Variable} \bullet \\
\quad ((c \in \operatorname{ran} \mathit{PositiveCondition} \Rightarrow \\
\qquad (\exists\, \mathit{wt} : \mathit{WMETest} \mid \mathit{wt} = \mathit{PositiveCondition}^{\sim}(c) \bullet \\
\qquad \mathit{ConditionsBranchingFactor}(c, \mathit{bound}, \mathit{positive}, \mathit{impasses}) = \\
\qquad\qquad \mathit{SBF}(\mathit{ClassifyWMETest}(\mathit{wt}, \mathit{bound}, \mathit{impasses}), \mathit{wt}))) \land \\
\quad (c \in \operatorname{ran} \mathit{NegativeCondition} \cup \operatorname{ran} \mathit{NegativeConjunctiveCondition} \Rightarrow \\
\qquad (((\mathit{ConditionsPositiveComponents}(c) \cap \mathit{Variable}) \cap \mathit{positive} \subseteq \mathit{bound}) \Rightarrow \\
\qquad\quad \mathit{ConditionsBranchingFactor}(c, \mathit{bound}, \mathit{positive}, \mathit{impasses}) = 1) \land \\
\qquad (\neg\, ((\mathit{ConditionsPositiveComponents}(c) \cap \mathit{Variable}) \cap \mathit{positive} \subseteq \mathit{bound}) \Rightarrow \\
\qquad\quad \mathit{ConditionsBranchingFactor}(c, \mathit{bound}, \mathit{positive}, \mathit{impasses}) = \mathit{Infinity})))
\end{array}
$$

D.3 The Reorderer's State

The reorderer's state machine uses three states:

1. *ReordererInitialState* — the machine starts in this state.
2. *ReordererAddConditionState* — the machine adds conditions in this state.
3. *ReordererFinishedState* — the machine finishes in this state.

$$
\begin{array}{rl}
\mathit{ReordererState} ::= & \mathit{ReordererInitialState} \\
\mid & \mathit{ReordererAddConditionState} \\
\mid & \mathit{ReordererFinishedState}
\end{array}
$$

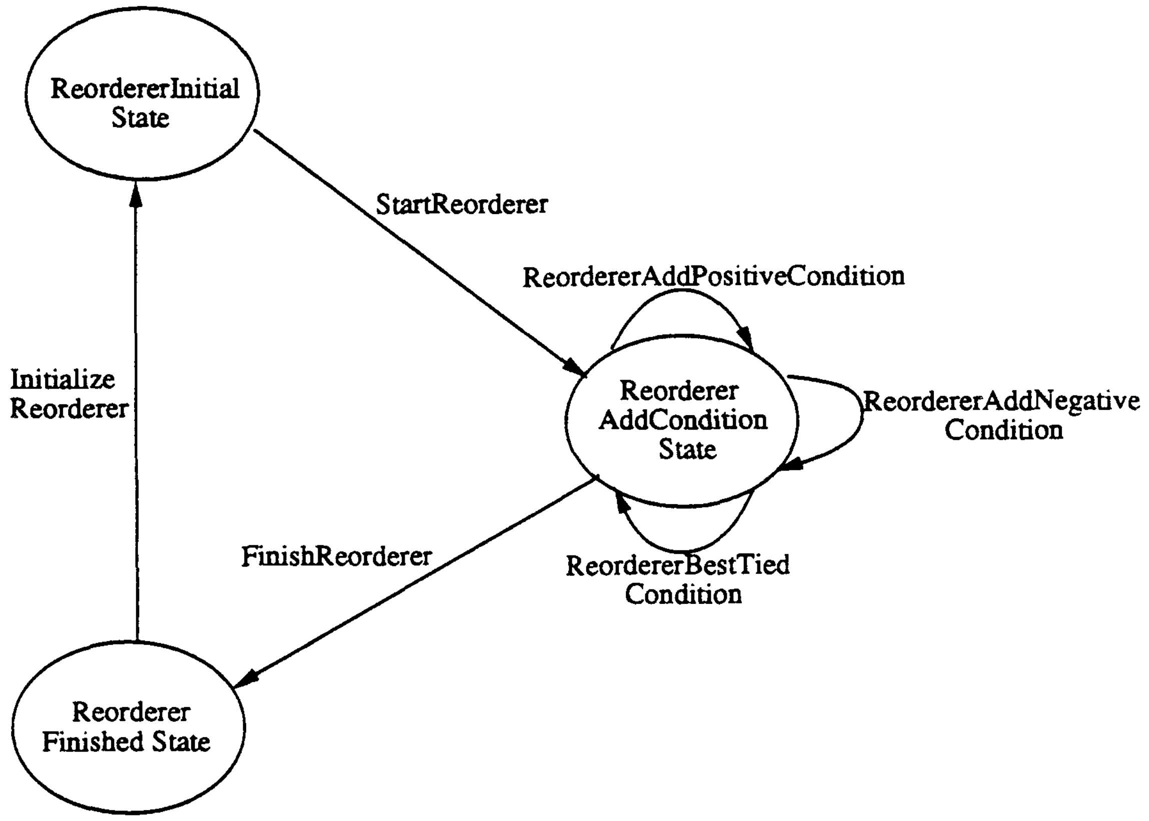
ReordererInitial State
StartReorderer
ReordererAddPositiveCondition
Initialize Reorderer
Reorderer AddCondition State
ReordererAddNegative Condition
FinishReorderer
ReordererBestTied Condition
Reorderer Finished State

Figure D.1: Reorderer State Machine

The *Reorderer* schema holds a state counter and seven pieces of state:

1. *ordering* — the sequence of conditions selected so far
2. *conditions* — the set of conditions of the production's LHS that have not been added to the ordering
3. *impasses* — the set of variables that the condition binds to goal and impasse identifiers
4. *bound* — the set of variables bound in a condition already in the ordering
5. *positive* — the set of variables bound in all of the positive conditions and
6. *branching_factor* — the estimated average cost of matching the entire ordering.

$$
\begin{array}{|l}
\hline
\textit{Reorderer} \\
\hline
reorderer\_state : ReordererState \\
ordering : \mathrm{seq}\ Condition \\
conditions : \mathbb{P}\ Condition \\
impasses, bound, positive : \mathbb{P}\ Variable \\
branching\_factor : \mathbb{N}_1 \\
\hline
\end{array}
$$

When the reorderer is initialized it must start with a set of conditions to order and an empty ordering. The branching factor of an empty ordering is defined to be one. The set of bound variables is empty because no conditions have been added to the ordering. Only goal or impasse conditions can be selected to start. so there must be some goal or impasse variables. The positive variables are the variables bound positively in the positive conditions.

$$
\begin{array}{|l}
\hline
\textit{InitReorderer} \\
\hline
Reorderer \\
\hline
conditions \neq \varnothing \\
ordering = \langle\rangle \\
bound = \varnothing \\
branching\_factor = 1 \\
impasses \neq \varnothing \\
positive = \\
\quad (\bigcup(ConditionsPositiveComponents\,(\!|\,conditions \cap \mathrm{ran}\ PositiveCondition\,|\!))) \\
\qquad \cap\ Variable \\
\hline
\end{array}
$$

When the reorderer's state is changed the root conditions and positive variables remain unchanged. With each step, the set of bound conditions grows and the branching factor is non-decreasing. The last predicate ensures that the reorderer either adds a condition to the end of ordering or leaves it unchanged.

$\Delta Reorderer$

$Reorderer$
$Reorderer'$

$impasses' = impasses$

$positive' = positive$

$bound \subseteq bound'$

$branching\_factor' \geq branching\_factor$

$ordering' = ordering \vee tail(ordering) = ordering'$

D.4 The Reorderer's Operations

InitializeReorderer moves the state machine from the finished state into the initial state and ensures that the reorderer's configuration is a valid initial configuration.

$InitializeReorderer$

$\Delta Reorderer$

$reorderer\_state = ReordererFinishedState$

$InitReorderer$

$reorderer\_state' = ReordererInitialState$

The machine's first action is to move to the add condition state.

$StartReorderer$

$\Delta Reorderer$

$reorderer\_state = ReordererInitialState$

$reorderer\_state' = ReordererAddConditionState$

The *BestConditions* function selects the subset of conditions that have a minimum branching factor.

$$
\begin{array}{l}
\mathit{BestConditions} : \\
\quad \mathbb{P}\ \mathit{Condition} \times \mathbb{P}\ \mathit{Variable} \times \mathbb{P}\ \mathit{Variable} \times \mathbb{P}\ \mathit{Variable} \rightarrow \mathbb{P}\ \mathit{Condition} \\
\hline
\forall\ C : \mathbb{P}\ \mathit{Condition};\ \mathit{bound}, \mathit{positive}, \mathit{impasses} : \mathbb{P}\ \mathit{Variable} \bullet \\
\quad \mathit{BestConditions}(C, \mathit{bound}, \mathit{positive}, \mathit{impasses}) = \\
\quad\quad \{c : C \mid \\
\quad\quad\quad \neg\ (\exists\ c_2 : C \bullet \\
\quad\quad\quad\quad\quad \mathit{ConditionsBranchingFactor}(c_2, \mathit{bound}, \mathit{positive}, \mathit{impasses}) < \\
\quad\quad\quad\quad\quad \mathit{ConditionsBranchingFactor}(c, \mathit{bound}, \mathit{positive}, \mathit{impasses}))\}
\end{array}
$$

When there is a unique best condition or the branching factor of all of the conditions is one, the reorderer will add a condition from the set of best conditions. The reorderer picks any one of the best conditions and adds it to the end of the ordering. It updates the estimate of the cost of the ordering by multiplying it by the condition's branching factor. The positive variables of the added condition are added into the set of bound variables.

$$
\begin{array}{l}
\textit{ReordererAddBestCondition} \\
\Delta \mathit{Reorderer} \\
\hline
\mathit{reorderer\_state} = \mathit{ReordererAddConditionState} = \mathit{reorderer\_state}' \\
\\
\#(\mathit{BestConditions}(\mathit{conditions}, \mathit{bound}, \mathit{positive}, \mathit{impasses})) = 1\ \vee \\
(\forall\ c : \mathit{BestConditions}(\mathit{conditions}, \mathit{bound}, \mathit{positive}, \mathit{impasses}) \bullet \\
\quad \mathit{ConditionsBranchingFactor}(c, \mathit{bound}, \mathit{positive}, \mathit{impasses}) = 1) \\
\\
\exists\ c : \mathit{BestConditions}(\mathit{conditions}, \mathit{bound}, \mathit{positive}, \mathit{impasses}) \bullet \\
\quad (\mathit{conditions}' = \mathit{conditions} \setminus \{c\}\ \wedge \\
\quad \mathit{ordering}' = \mathit{ordering} \frown \langle c \rangle\ \wedge \\
\quad \mathit{branching\_factor}' = \quad \mathit{branching\_factor} * \\
\quad\quad\quad \mathit{ConditionsBranchingFactor}(c, \mathit{bound}, \mathit{positive}, \mathit{impasses})\ \wedge \\
\quad \mathit{bound}' = \mathit{bound} \cup (\mathit{ConditionsPositiveComponents}(c) \cap \mathit{Variable}))
\end{array}
$$

LookAhead is given: a condition, the set of all conditions yet to be chosen, the bound variables, the positive variables and the goal or impasse variables. It calculates the branching factor for first adding the condition and then adding the best of the remaining conditions.

$$\begin{array}{l}
\mathit{LookAhead} : \\
\quad \mathit{Condition} \times \mathbb{P}\ \mathit{Condition} \times \mathbb{P}\ \mathit{Variable} \times \mathbb{P}\ \mathit{Variable} \times \mathbb{P}\ \mathit{Variable} \rightarrow \mathbb{Z} \\
\hline
\forall c_1 : \mathit{Condition};\ C : \mathbb{P}\ \mathit{Condition};\ \mathit{bound}, \mathit{positive}, \mathit{impasses} : \mathbb{P}\ \mathit{Variable} \mid c_1 \in C \bullet \\
\quad \exists\ \mathit{newbounds} : \mathbb{P}\ \mathit{Variable} \mid \\
\qquad \mathit{newbounds} = \mathit{bound} \cup (\mathit{ConditionsPositiveComponents}(c_1) \cap \mathit{Variable}) \bullet \\
\qquad \exists c_2 : \mathit{BestConditions}(C \setminus \{c_1\}, \mathit{bound}, \mathit{positive}, \mathit{impasses}) \bullet \\
\qquad\quad \mathit{LookAhead}(c_1, C, \mathit{bound}, \mathit{positive}, \mathit{impasses}) = \\
\qquad\qquad \mathit{ConditionsBranchingFactor}(c_1, \mathit{bound}, \mathit{positive}, \mathit{impasses}) * \\
\qquad\qquad \mathit{ConditionsBranchingFactor}(c_2, \mathit{newbounds}, \mathit{positive}, \mathit{impasses})
\end{array}$$

BestTiedConditions is used to break ties between conditions. It returns the set of conditions that have the minimum branching factor after one step of lookahead.

$$\begin{array}{l}
\mathit{BestTiedConditions} : \\
\quad \mathbb{P}\ \mathit{Condition} \times \mathbb{P}\ \mathit{Variable} \times \mathbb{P}\ \mathit{Variable} \times \mathbb{P}\ \mathit{Variable} \rightarrow \mathbb{P}\ \mathit{Condition} \\
\hline
\forall C : \mathbb{P}\ \mathit{Condition};\ \mathit{bound}, \mathit{positive}, \mathit{impasses} : \mathbb{P}\ \mathit{Variable} \bullet \\
\quad \mathit{BestTiedConditions}(C, \mathit{bound}, \mathit{positive}, \mathit{impasses}) = \\
\qquad \{c_1 : \mathit{BestConditions}(C, \mathit{bound}, \mathit{positive}, \mathit{impasses}) \mid \\
\qquad\quad \neg\,(\exists c_2 : \mathit{BestConditions}(C, \mathit{bound}, \mathit{positive}, \mathit{impasses}) \bullet \\
\qquad\qquad \mathit{LookAhead}(c_2, C \setminus \{c_1\}, \mathit{bound}, \mathit{positive}, \mathit{impasses}) < \\
\qquad\qquad\quad \mathit{LookAhead}(c_1, C \setminus \{c_1\}, \mathit{bound}, \mathit{positive}, \mathit{impasses}))\}
\end{array}$$

Whenever there is a tie for the best condition and all of the tied conditions have a branching factor of greater than one, *ReordererAddBestTiedCondition* selects one of the best tied conditions and adds it to the ordering.

```
— ReordererAddBestTiedCondition ————————————
  ΔReorderer
————————————
  reorderer_state = ReordererAddConditionState = reorderer_state'

  ¬ (#(BestConditions(conditions, bound, positive, impasses)) = 1 ∨
  (∃ c : BestConditions(conditions, bound, positive, impasses) •
     ConditionsBranchingFactor(c, bound, positive, impasses) = 1))

  ∃ c : BestTiedConditions(conditions, bound, positive, impasses) •
    (conditions' = conditions \ {c} ∧
     ordering' = ordering ⁀ ⟨c⟩ ∧
     branching_factor' =
         branching_factor *
               ConditionsBranchingFactor(c, bound, positive, impasses) ∧
     bound' = bound ∪ (ConditionsPositiveComponents(c) ∩ Variable))
————————————
```

When all of the conditions have been added to the ordering, *FinishReorderer* moves the reorderer into the finished state.

```
— FinishReorderer ————————————
  ΔReorderer
————————————
  reorderer_state = ReordererAddConditionState

  conditions = ∅

  reorderer_state' = ReordererFinishedState
————————————
```

StepReorderer steps the reorderer.

```
StepReorderer ≙ StartReorderer ∨
        ReordererAddBestCondition ∨ ReordererAddBestTiedCondition ∨
        FinishReorderer
```

ResetReorderer allows the system to reset the reorderer to the finished state from any state.

```
— ResetReorderer ————————————
  ΔReorderer
————————————
  reorderer_state' = ReordererFinishedState
————————————
```

For simplicity, the reordering of conjunctive negations has not been specified. Whenever a conjunctive negation is added to the ordering the system should reorder the conditions of the negation using the same algorithm. It should start with: an empty ordering, the bound variables of the entire ordering, the roots of the entire ordering and the positively tested variables of the positive conditions of the conjunctive negation.

Bibliography

[Aky90] A. Akyurek. Means-Ends Planning, Operator Subgoaling, and Operator Valuation: An Example Soar Program. Technical Report RUG-FA-90-3, University of Groningen, August 1990.

[BW90] Michael Barr and Charles Wells. *Category Theory for Computing Science.* Prentice Hall International, 66 Woodlane End, Hemmel Hampstead, Hertfordshire HP24RG UK, 1990.

[Cra91] Iain D Craig. *Formal Specification of Advanced AI Architectures.* Ellis Horwood Limited, Market Cross House, Cooper Street, Chichester, West Sussex, PO19 1EB, England, 1991.

[DG89] N Delisle and D Garlan. Formally specifying electronic instruments. In *Proceedings of Fifth International Workshop on Software Specification and Design*, pages 242–248, 19-20 May 1989.

[DS90] M. DeJongh and Jr. Smith, J. W. Blood Typing: Functional Modeling Put to the Tests. Laboratory for Artificial Intelligence Research, Ohio State University, November, 1990, Unpublished., 1990.

[Gar91] David Garlan. Formalizing design spaces: implicit invocation mechanisms. Technical Report CMU-CS-91-114, Carnegie Mellon University, School of Computer Science, 1991.

[Gut90] John V. Guttag. Report on the Larch Shared Language. Technical Report 58, Digital Equipment Corporation, Systems Research Center, 1990.

[GW88] Joseph A Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Laboratory, 1988.

[Hay85] Ian Hayes, editor. *Specification Case Studies.* Prentice Hall International, 66 Woodlane End, Hemmel Hampstead, Hertfordshire HP24RG UK, 1985.

[HPS89] W. Hsu, M. Prietula, and D.M. Steier. Merl-Soar: Scheduling within a general architecture for intelligence. In *Proceedings of the Third International Conference on Expert Systems and the Leading Edge in Production and Operations Management*, pages 467–481, May 1989.

[Jon90] C.B. Jones. *Systematic Software Development using VDM (2nd edition)*. Prentice Hall International, 66 Woodlane End, Hemmel Hampstead, Hertfordshire HP24RG UK, 1990.

[JRS91] B.E. John, R.W. Remington, and D.M. Steier. An Analysis of Space Shuttle Countdown Activities: Preliminaries to a Computational Model of the NASA Test Director. Technical Report CMU-CS-91-138, School of Computer Science, Carnegie Mellon University, May 1991.

[JS90] C. B. Jones and R. (editors) Shaw. *Case Studies in VDM*. Prentice Hall International, 66 Woodlane End, Hemmel Hampstead, Hertfordshire HP24RG UK, 1990.

[KR88] Brian W. Kernighan and Dennis M. Richie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1988.

[Lai86] J.E. Laird. Soar User's Manual: Version 4.0. Technical report, Intelligent Systems Laboratory, Palo Alto Research Center, Xerox Corporation, January 1986. Out of Print, see the Version 5.2 manual.

[LCAS90] J.E. Laird, C.B. Congdon, E. Altmann, and K. Swedlow. Soar User's Manual: Version 5.2. Technical report, Electrical Engineering and Computer Science, University of Michigan, October 1990. Also available from The Soar Group, School of Computer Science, Carnegie Mellon University, CMU-CS-90-179.

[LLN91] J. Fain Lehman, R.L. Lewis, and A. Newell. Integrating Knowledge Sources in Language Comprehension. School of Computer Science, Carnegie Mellon University, January 1991, Unpublished, 1991.

[LNR87] J.E. Laird, A. Newell, and P.S Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[LRN84] J.E. Laird, P.S. Rosenbloom, and A. Newell. Towards chunking as a general learning mechanism. In *Proceedings of the National Conference on Artificial Intelligence*, pages 188–192, August 1984.

[LRN86] J. E. Laird, P.S. Rosenbloom, and A. Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[LYHT91] J.E. Laird, E.S. Yager, M. Hucka, and C.M. Tuck. Robo-Soar: An integration of external interaction, planning, and learning using Soar. In Van de Velde, editor, *Machine Learning for Autonomous Agents*. MIT Press: Bradford Books, Cambridge, Massachusetts, 1991.

[New90] A. Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1990.

[NYL+91] A. Newell, G.R. Yost, Laird, P.S. J.E., Rosenbloom, and E. Altmann. Formulating the problem space computational model. In R.F. Rashid, editor, *Carnegie Mellon Computer Science. A 25-Year Commerative*, pages 255–293. ACM-Press: Addison-Wesley, Reading, PA, 1991.

[PMN91] G. Pelton, B. G. Milnes, and A. Newell. NSoar External Interface Informal Specification, 1991. NSoar Group, School of Computer Science, Carnegie Mellon University, May, 1991, Unpublished.

[PTS91] Ben Potter, David Till, and Jane Sinclair. *An Introduction to formal specification and Z*. Prentice Hall International, 66 Woodlane End, Hemmel Hampstead. Hertfordshire HP24RG UK, 1991

[RL86] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto soar. In *Proceedings of the National Conference on Artificial Intelligence*, pages 561–567, August 1986.

[Sca86] D.J. Scales. Efficient matching algorithms for the SOAR/OPS5 production system. Technical Report KSL-86-47, Knowledge Systems Laboratory, Stanford University, June 1986.

[SK91] A. Simon, T.and Newell and D. Klahr. Q-Soar: A computational account of children's learning about number conservation. In *Working Models of Human Perception*. Morgan Kaufman, Los Altos, California, 1991.

[Spi88] J.M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, Cambridge, Cambridgeshire, UK, 1988.

[Spi89] J.M. Spivey. *The Z Notation*. Prentice Hall, Hertfordshire, UK, 1989.

[Tam91] M. Tambe. *Eliminating Combinatorics from Production Match*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.

[Wor92] J. B. Wordsworth. *Software Development with Z* International Computer Science series. Addison Wesley, 1992. ISBN 0-210-62757-4.

List of Definitions